Final Project:

Facebook V: Predicting Check Ins

Definition

Project Overview

This is a project from one of Kaggle's ongoing competition projects. It's a machine learning engineering competion provided by Facbook. I have little experience in doing Kaggle competitions. To me, this project is a challenge, but as well as a chance. I want to use this project to sharpen what I've learned from Udacity.

We know the enormous value and pontential of predicting people's behavior. Such a prediction could be based on the detection of a broad range of features. A typical example is Netflix's recommendation system. It recommends programs mainly based on one's watching history.

This project is interesting. It's also trying to predict people's behavior. To be specific, its goal is to predict more accurately which places one is likely to check in to, based on time and the inaccurate detection of one's location. In such a way, Facebook maybe able to provide better user experience. Although the collected data itself is not actual data but from an artificial world created by Facebook, it has been fabricated to resemble data from a real world.

The data sets provided by facebook are pretty simple: training data, testing data and an example of what to submit. After training and testing my model, I also need to create my predictions and generate a .csv file to submit to the online judge.

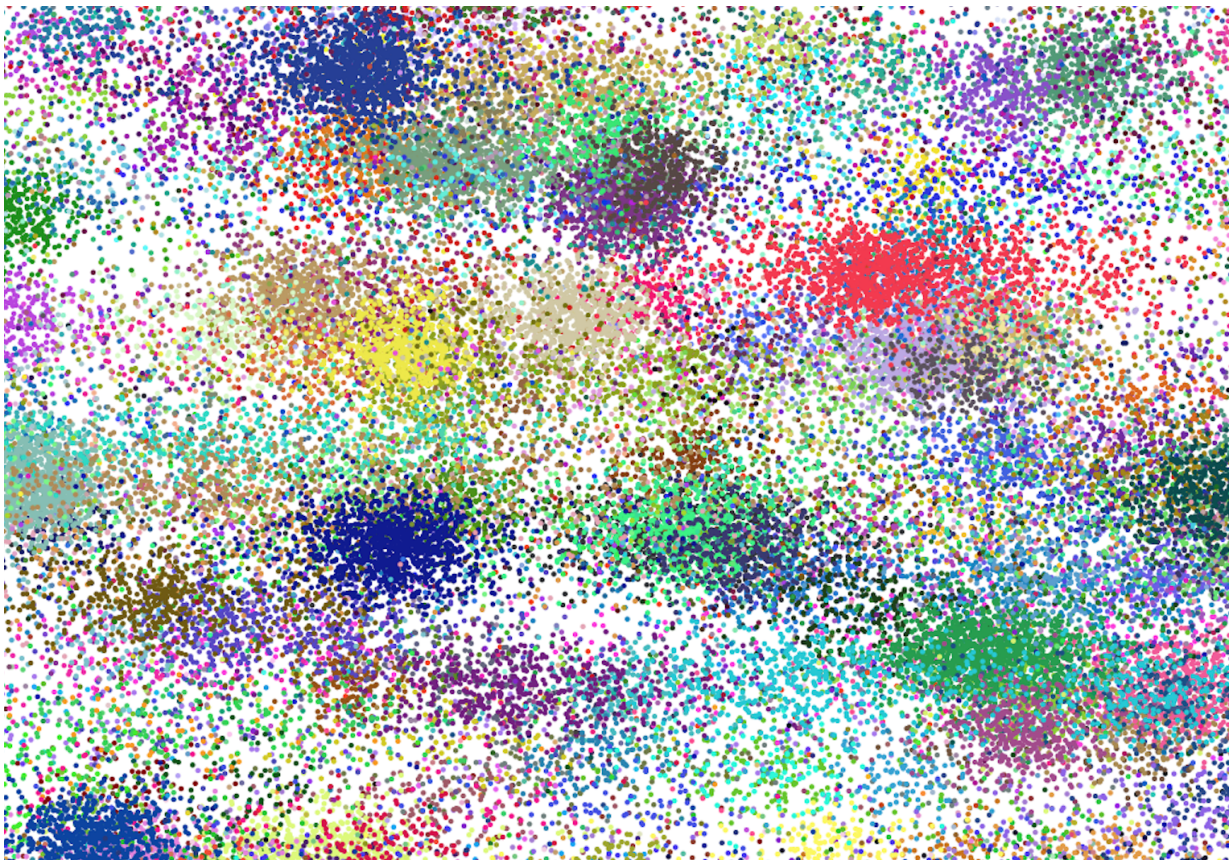
Project Statement

The official statement of this project is following:

The goal of this competition is to predict which place a person would like to check in to. For the purposes of this competition, Facebook created an artificial world consisting of more than 100,000 places located in a 10 km by 10 km square. For a given set of coordinates, your task is to return a ranked list of the most likely places. Data was fabricated to resemble location signals coming from

mobile devices, giving you a flavor of what it takes to work with real data complicated by inaccurate and noisy values. Inconsistent and erroneous location data can disrupt experience for services like Facebook Check In.

In this competition, you are going to predict which business a user is checking into based on their location, accuracy, and timestamp.



The train and test dataset are split based on time, and the public/private leaderboard in the test data are split randomly. There is no concept of a person in this dataset. All the row_id's are events, not people.

For every user check in, you must predict a space-delimited list of the businesses they check into. You may submit up to 3 predictions for each check in. The file should contain a header and have the following format:

```
row_id,place_id

0,2083653582 1476539553 1000015801

1,6147316961 6147316961 8999137193

etc...
```

From the above official statement, we can see this is a classification problem. There are a few input features. However, we can see from the above figure, the complexity is the large number of labels. This classification problem does not have True/False labels but instead has a lot of classes. I will try different classification algortithms and find one that provides best predictions.

Metrics

The following is the official evaluation metric:

Submissions are evaluated according to the Mean Average Precision @3 (MAP@3):

$$\mathrm{MAP}\ @\ 3 = \frac{1}{|U|}\sum_{u=1}^{|U|}\sum_{k=1}^{\min(3,n)} \mathrm{P}(k)$$

where |U| is the number of check in events, P(k) is the precision at cutoff k, n is the number of predicted businesses.

In the statement, I'm required to submit 3 predictions according to their importance, for each test entry. This metric decides how close the prediction is to the right answer. Enven if one prediction missed the shot, the precision is not zero because other predictions may be right. The score does not only estimate how many outputs hitting the answer, the sequence of these outputs also matters.

This kind of metric is commonly used in the area of recommendation.

Analysis

Data Exploration

First, let's load the data.

In [3]:
```
%matplotlib inline
import numpy as np
import pandas as pd
from ggplot import *
import matplotlib.pyplot as plt
import matplotlib.pylab as pylab
import random
```

In [4]:
```
data_train = pd.read_csv('train.csv')
```

The train.csv data is the data we can use to train or test our model. There is also a test.csv file which contains the data we use to do the prediction. Now, let's have a glance at the format of the training data set.

In [5]:
```
data_train.head()
```

Out[5]:

	row_id	x	y	accuracy	time	place_id
0	0	0.7941	9.0809	54	470702	8523065625
1	1	5.9567	4.7968	13	186555	1757726713
2	2	8.3078	7.0407	74	322648	1137537235
3	3	7.3665	2.5165	65	704587	6567393236
4	4	4.0961	1.1307	31	472130	7440663949

There are 4 features (not including "row_id") with 1 label. The official statement explains the features and label as follows:

> row_id: id of the check-in event
>
> x y: coordinates
>
> accuracy: location accuracy
>
> time: timestamp
>
> place_id: id of the business, **this is the target you are predicting**

Now let's do some foundmental statistics.

In [137]:
```
data_train.describe()
```

Out[137]:

	row_id	x	y	accuracy	time
count	29118021.000000	29118021.000000	29118021.000000	29118021.000000	29118021.
mean	14559010.000000	4.999770	5.001814	82.849125	417010.36
std	8405648.775647	2.857601	2.887505	114.751772	231176.14
min	0.000000	0.000000	0.000000	1.000000	1.000000
25%	7279505.000000	2.534700	2.496700	27.000000	203057.00
50%	14559010.000000	5.009100	4.988300	62.000000	433922.00
75%	21838515.000000	7.461400	7.510300	75.000000	620491.00
max	29118020.000000	10.000000	10.000000	1033.000000	786239.00

There are 29118021 rows, a huge number. "x" and "y" range from 0 to 10, means the city is 10 km by 10 km. The "accuracy"s unit is not known. My guess is that it has a unit of "feet". It could not be in Kilometers because the max value would exceed the 10 km X 10 km square. It's not very likely to be in "meter". The median value is 62, and that value seems to be too large for a morden GPS phone if the unit is meter. The normal GPS accuracy measured on a phone is 5 to 10 meters. Considering people are usually walking when using a app, this distance uncertainty could be

larger. So I think "feet" is the only reasonable unit for this feature. The unit of "time" is not known either. It's probably in minutes. I will discuss what I did to get this conclusion in later discussion. What we are especially interested is how many "place_id"s there are.

In [138]:
```
print "There are {} unique places in the virtual city.".format(len(data_tr
```

```
There are 108390 unique places in the virtual city.
```

What a big number! It is huger for a classification problem.

Now let's check if there is any missing value.

In [139]:
```
data_train.isnull().sum()
```

Out[139]:
```
row_id      0
x           0
y           0
accuracy    0
time        0
place_id    0
dtype: int64
```

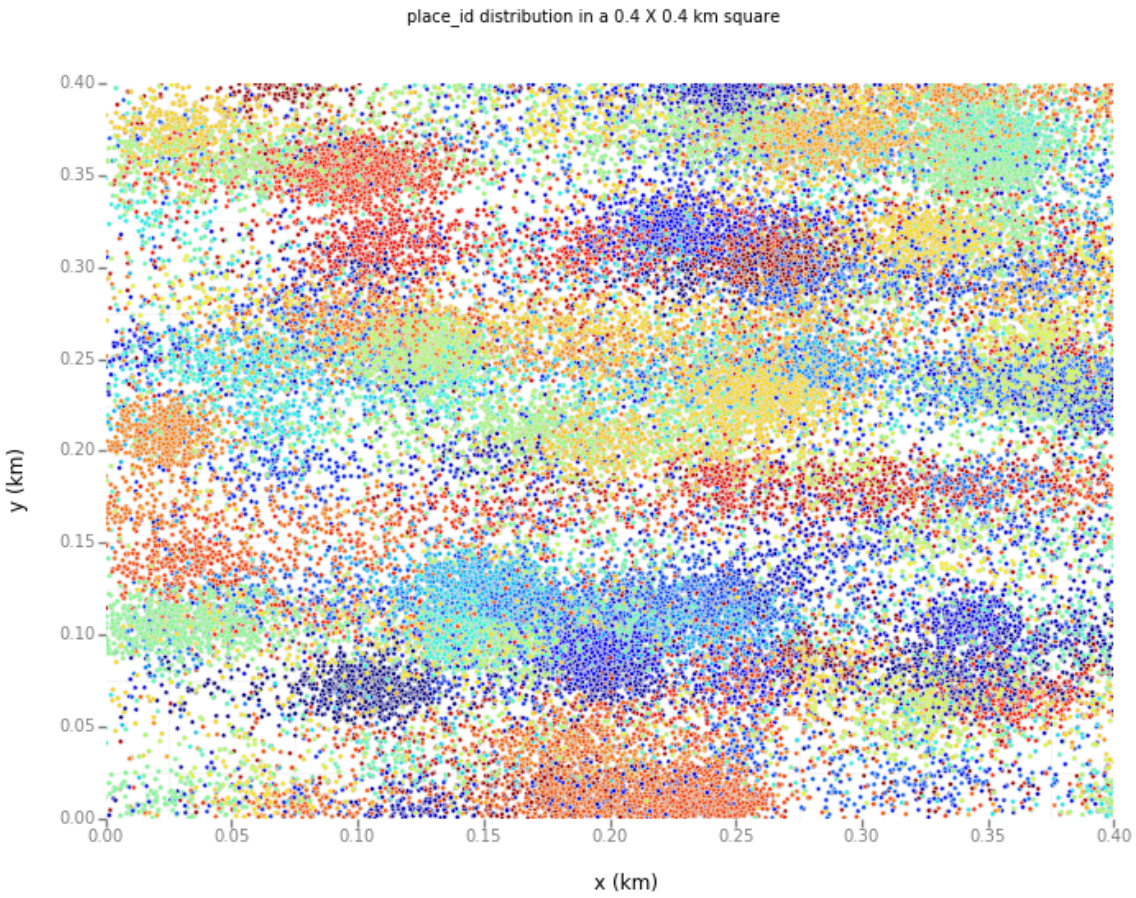
So there is no missing cells in the data set. That's good. We can go to the next step.

Exploratory Visualization

First let's see the locations of different "place_id"s or the clusterring.

The code in the following cell is limited to show the locations of "place_id"s in a small square of 0.4 X 0.4 km. I did not choose to show the locations in the whole city. One reason is that it would take a lot of time to do the calculation. The other reason is that one can not see clearly the clustering of individual points. With this smaller square, we can see points are distributed in such a way: points with the same label tends to form a cluster. This makes sense because same label means the individuals check in to the same place. The ideal case is that points with the same label overlap in the map. But due to the uncerterty of the location information, points can not overlap perfectly.

In [6]:
```
data_train_part = data_train[(data_train['x'] < 0.4) & (data_train['y'] <

ggplot(aes(x='x', y='y'), data=data_train_part) + geom_point(color = data_
    scale_x_continuous(limits=(0,0.4))  +\
    scale_y_continuous(limits=(0,0.4))  +\
    labs( x = "x (km)", y = "y (km)", title = "place_id distribution in a
    theme_bw()
```



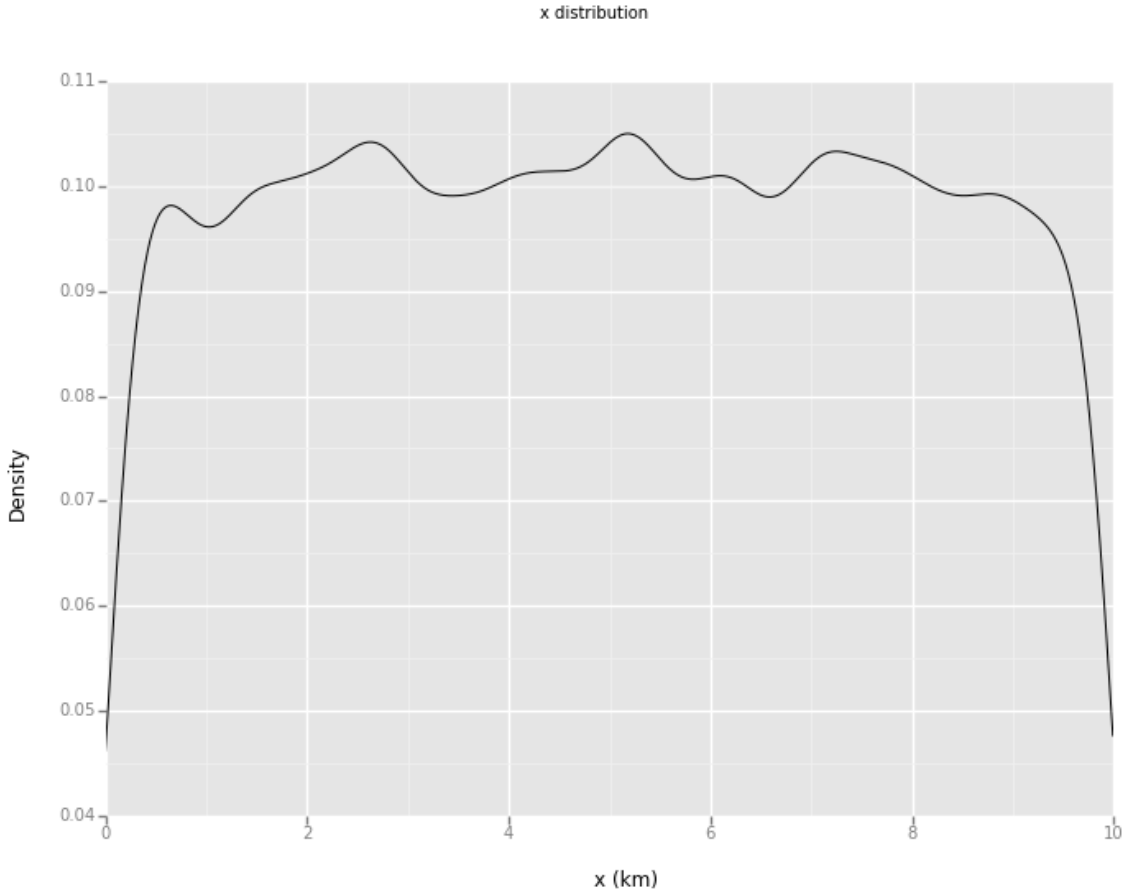
Out[6]: <ggplot: (274700641)>

There a many clusters in this graph. One thing I noticed interesting is that the height of a cluster is roughly half of the length.

Now let's see the distribution of each feature. I've tried to plot the whole data but it took too long to do the calculation. So in the following cell, I select randomly 200000 rows from the complete data and draw the distribution of each feature.

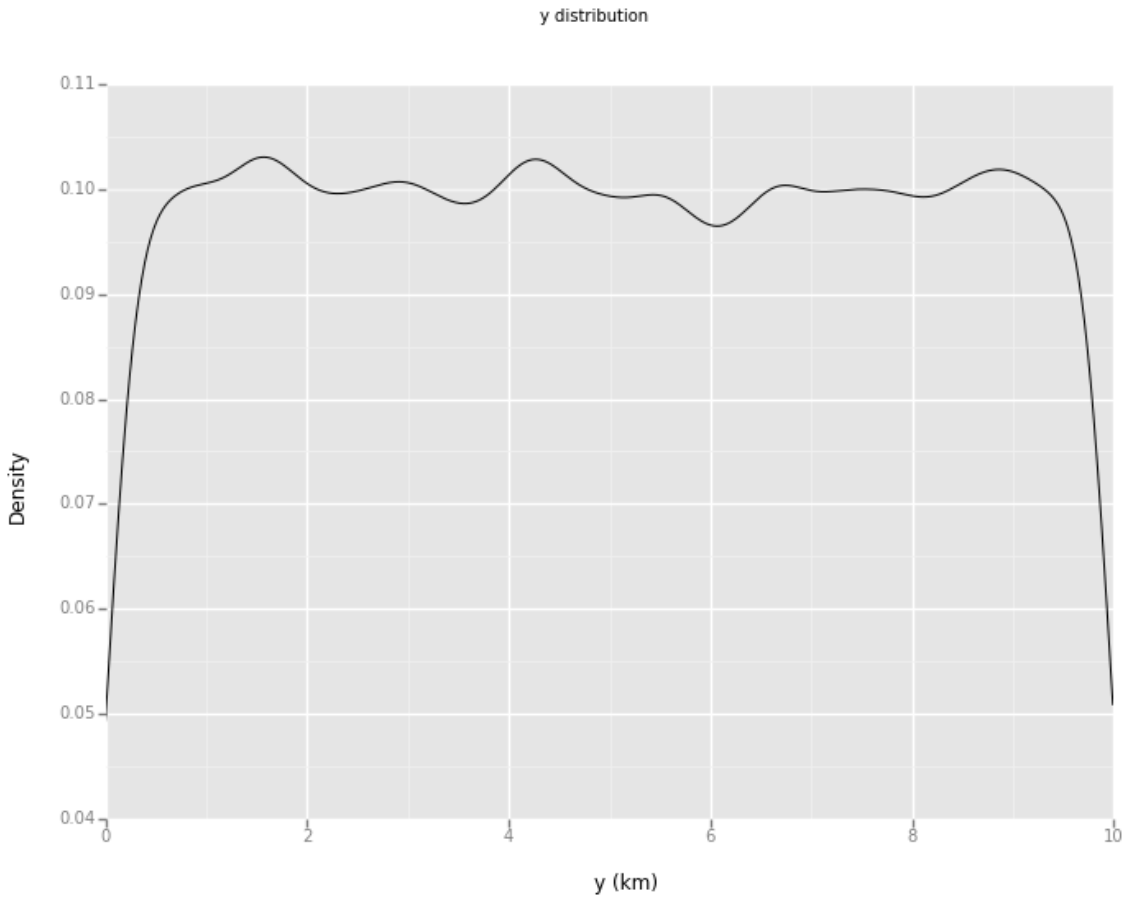
In [18]:
```
rows = np.random.choice(data_train.index.values, 200000)
data_train_sample = data_train.ix[rows].copy()
del rows
```

In [9]:
```
ggplot(aes(x = 'x'), data = data_train_sample) +\
    geom_density() +\
    labs( x = "x (km)", y = "Density", title = "x distribution")
```



Out[9]: <ggplot: (274700737)>

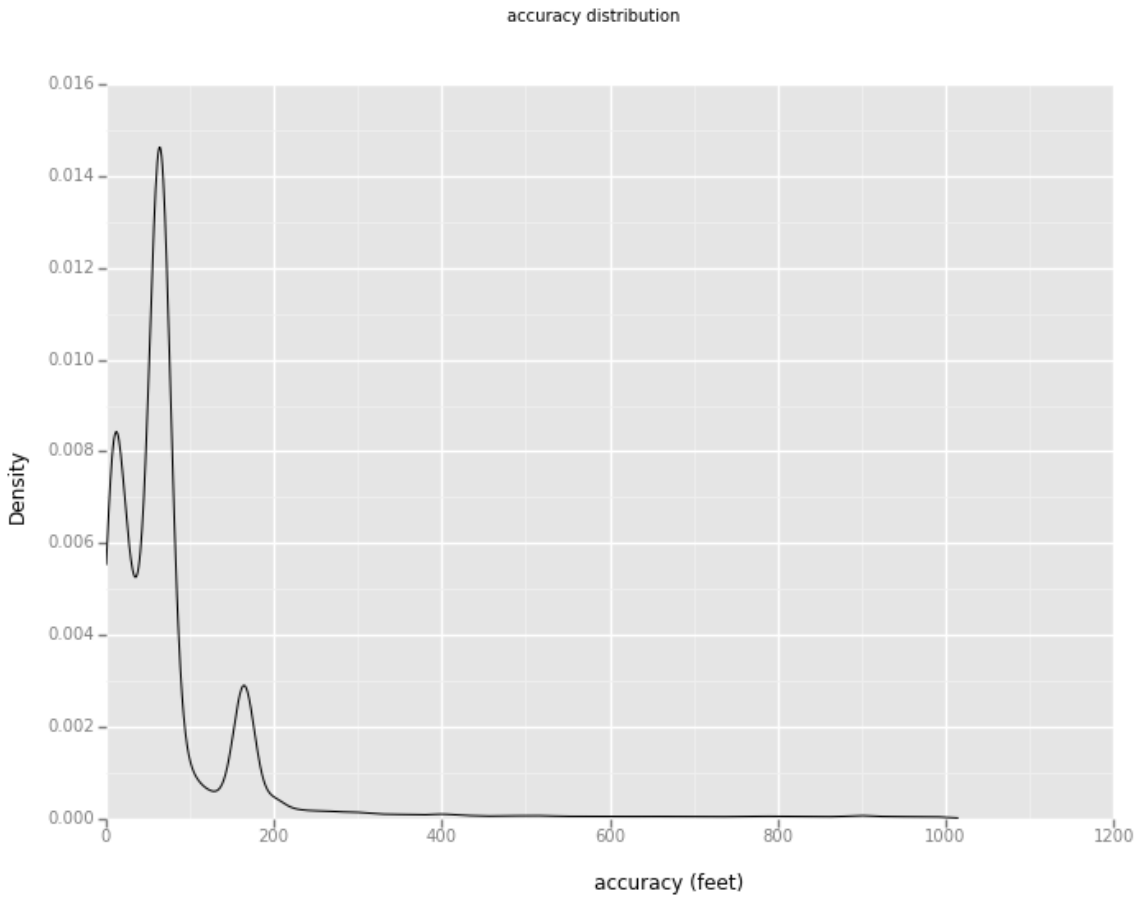
In [10]:
```
ggplot(aes(x = 'y'), data = data_train_sample) +\
    geom_density() +\
    labs( x = "y (km)", y = "Density", title = "y distribution")
```



Out[10]: <ggplot: (277034765)>

From the above two plots of x distribution and y distribution, we can see the density is small at the edge of the city but is roughly uniform for other places. That means random selection works well and can be representative for the whole data set.

In [11]:
```
ggplot(aes(x = 'accuracy'), data = data_train_sample) +\
    geom_density() +\
    labs( x = "accuracy (feet)", y = "Density", title = "accuracy distribu
```

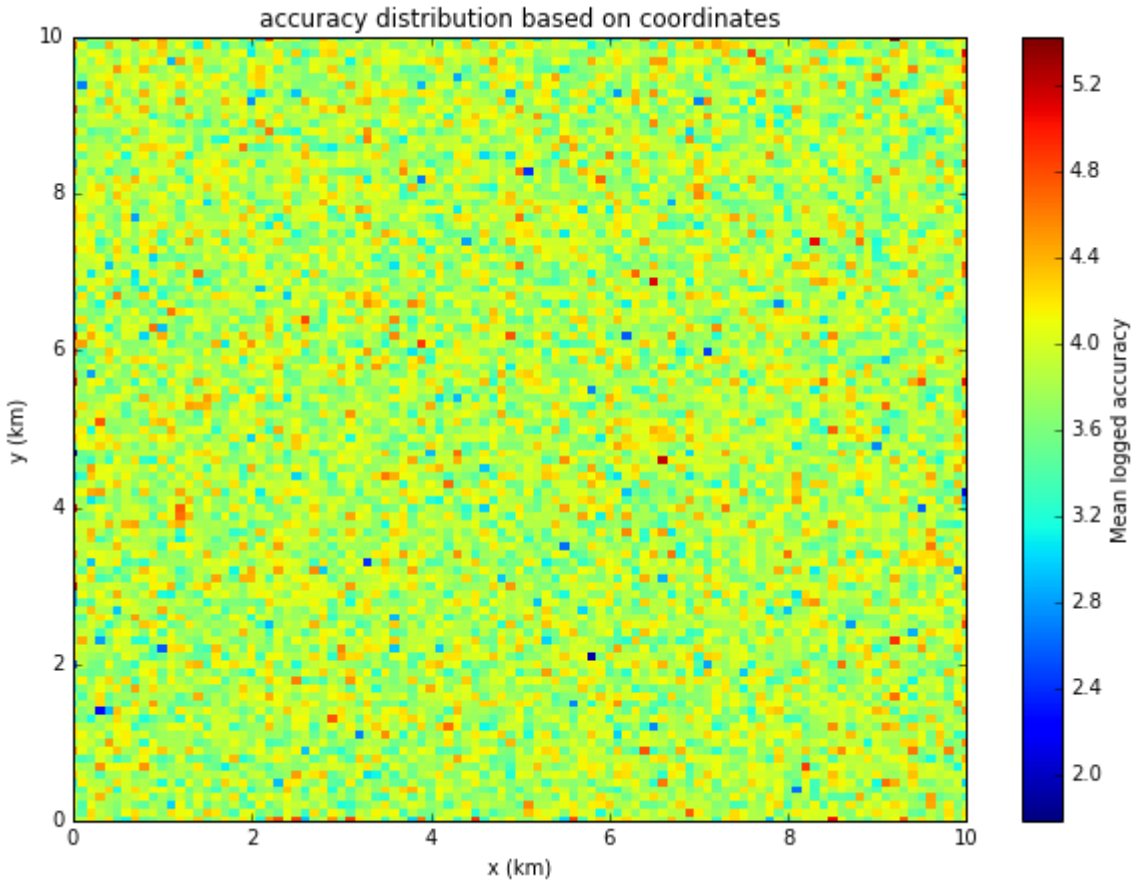


Out[11]: <ggplot: (274700705)>

The accuracy distribution plot is interesting. It is not a normal distribution as I expected. Neither is it a skewed distribution. It has a long tail and three peaks in the front part. One thought is that the accuracy may be affected by the x and y coordinates. Another thought is that there is a correlation between time and accuracy. Let's check these thoughts respectively.

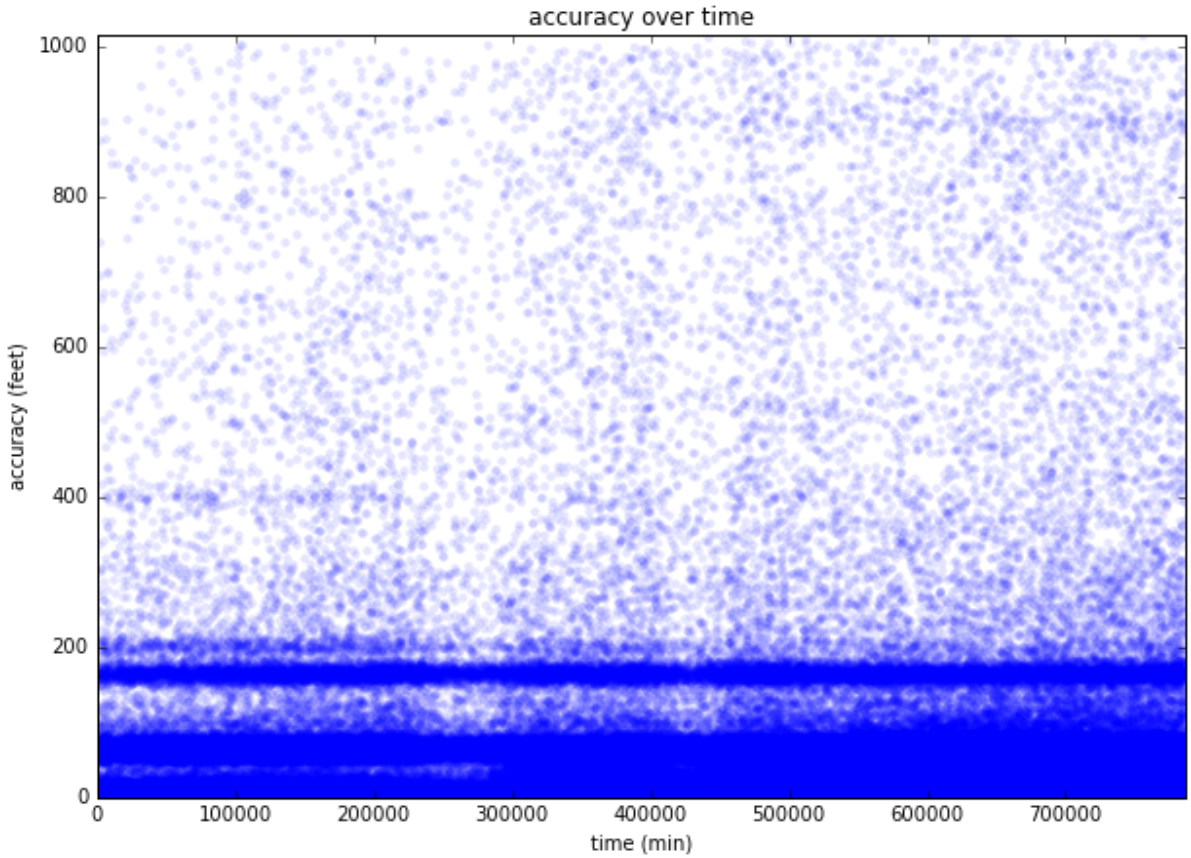
First, let's see how the accuracy changes with coordinates.

In [24]:
```
data_train_sample["xround"] = data_train_sample["x"].round(decimals=1)
data_train_sample["yround"] = data_train_sample["y"].round(decimals=1)
data_train_sample["accuracy_log"] = np.log(data_train_sample["accuracy"])
data_groupxy = data_train_sample.groupby(["xround", "yround"]).agg({"accu

idx = np.asarray(list(data_groupxy.index.values))

plt.figure(figsize=(10,7))
plt.scatter(idx[:,0], idx[:,1], c=data_groupxy["accuracy_log"], marker='s'
plt.colorbar().set_label("Mean logged accuracy")
plt.xlabel("x (km)")
plt.ylabel("y (km)")
plt.xlim(0,10)
plt.ylim(0,10)
plt.title("accuracy distribution based on coordinates")

plt.show()
```



It seems the accuracy distributes randomly in the map. Now, let's check how it changes with time.

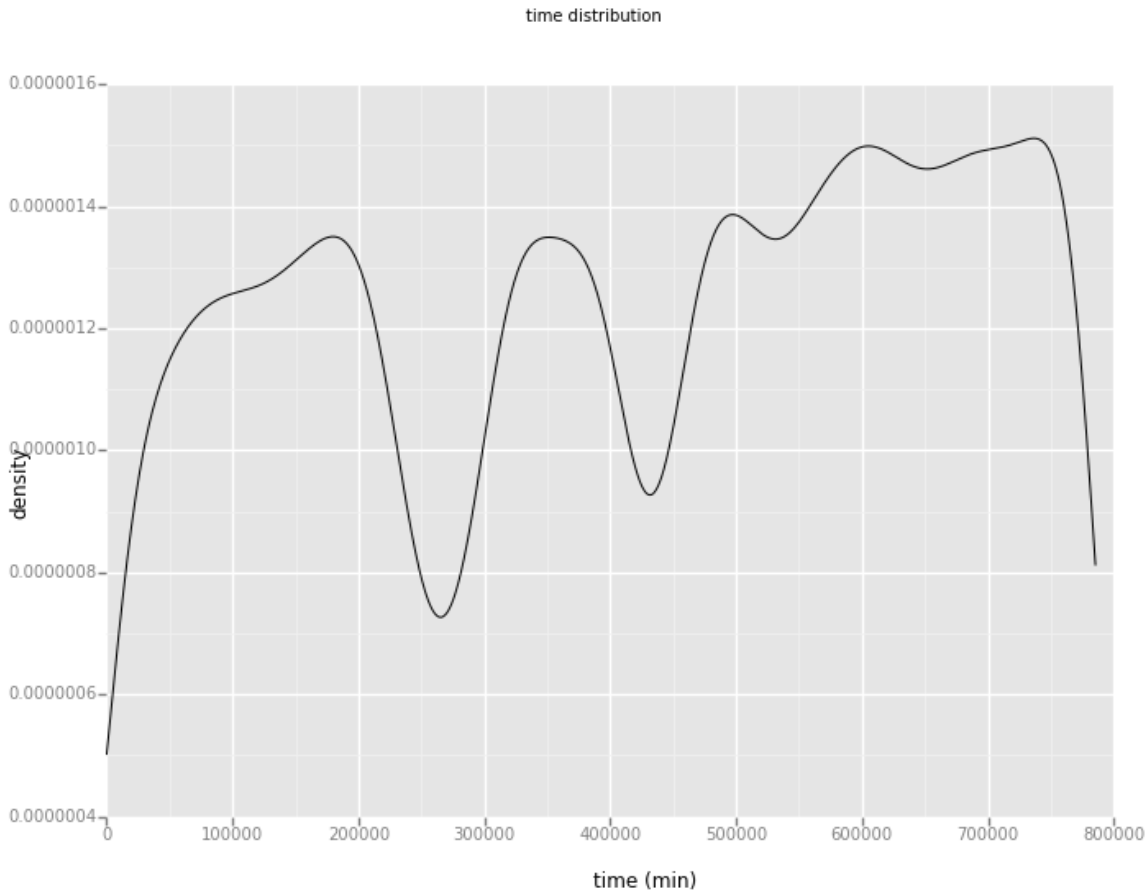
In [13]:
```
plt.figure(figsize=(10,7))
plt.scatter(data_train_sample["time"], data_train_sample["accuracy"], colo
plt.xlabel("time (min)")
plt.ylabel("accuracy (feet)")
plt.xlim(0, np.max(data_train_sample["time"]))
plt.ylim(0, np.max(data_train_sample["accuracy"]))
plt.title("accuracy over time")

plt.show()
```



The above figure still does not explain why the accuracy is not the common normal or uniform distribution. The strange distribution of accuracy must be due to some hiden features. I guess it's probably due to the features of the GPS models in phones. It has nothing to do with the features in this problem.

Now let's continue to check the "time" distribution.

In [14]:
```
ggplot(aes(x = 'time'), data = data_train_sample) +\
    geom_density() +\
    labs( x = "time (min)", y = "density", title = "time distribution")
```



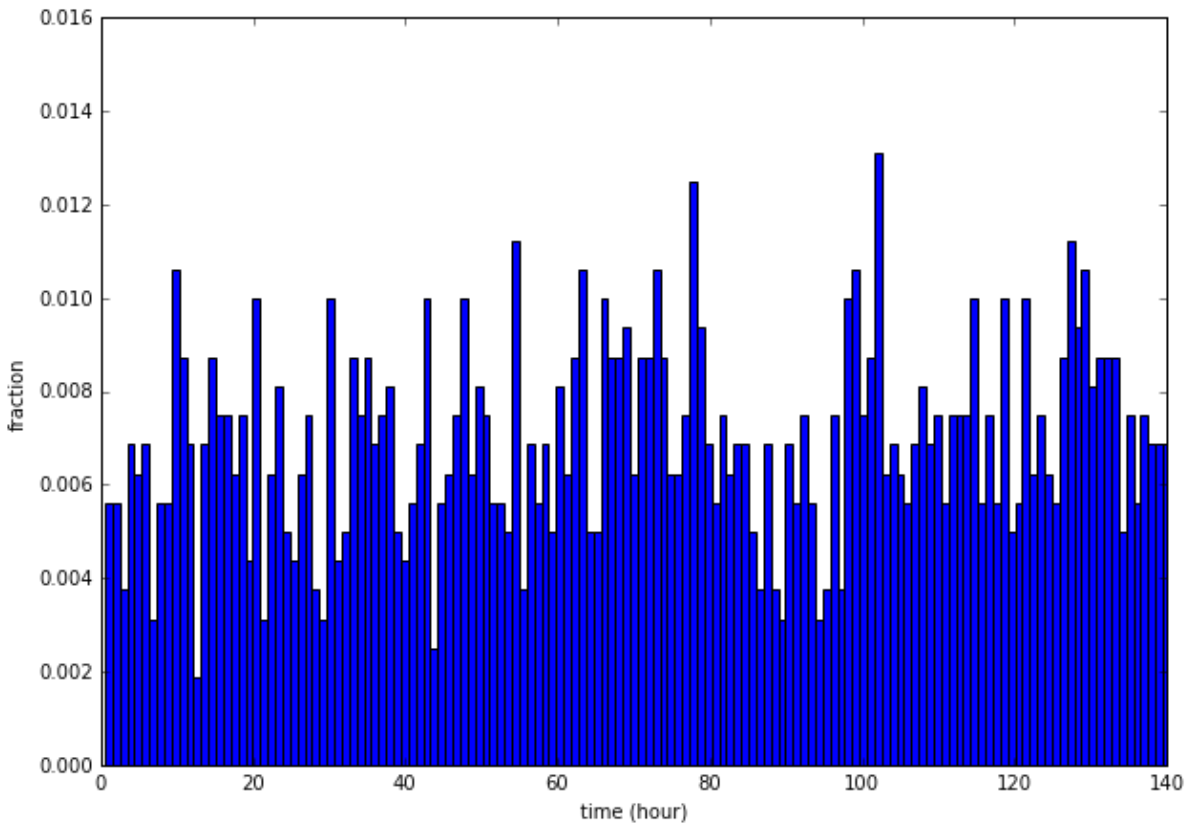
Out[14]: <ggplot: (280213021)>

The time distribution is also interesting. It has two big dips. If we check the figure of accuracy over time, we can also observe two obscure white strips. That means during some time periods, the number of people sending mobile signals to facebook has droped a lot.

In the above plot I already marked the time is in unit "min". I did this because I have already done the following reasoning part. Now let me show the reasoning.

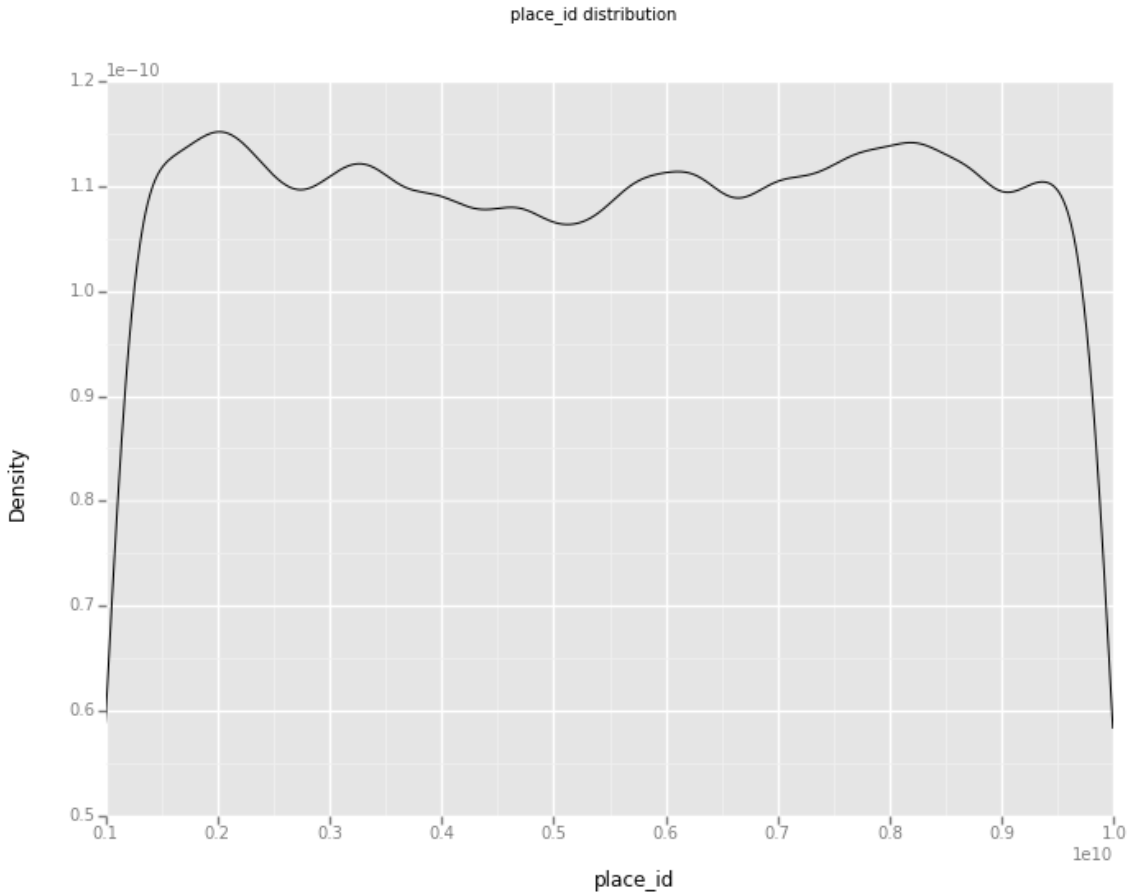
In [19]:
```
time_data_train = data_train_sample[data_train_sample["time"] < 1440 * 6]
counts, bins = np.histogram(time_data_train, range=[0, 140], bins=24 * 6)
binsc = bins[:-1] + np.diff(bins)/2.

plt.figure(figsize=(10,7))
plt.bar(binsc, counts/(counts.sum()*1.0), width=np.diff(bins)[0])
plt.axis([0, 140, 0, 0.016])
plt.xlabel("time (hour)")
plt.ylabel("fraction")
```

Out[19]: <matplotlib.text.Text at 0x107255a90>



We can look major peaks. By counting the space between two nearby major peaks, we can find they are separated by around 24 bins. Supposing the time is unit "minute", that is 24 hours, which is a cycle of a day. This makes sense. If the unit is "second", "hour" or "day", the regular reappearance of big peaks could not be explained. Moveover, it also explains why there are two big dips in the "time" distribution. The separation between the two big dips is around 170000 minutes, which is one week.

In [22]:
```
ggplot(aes(x = 'place_id'), data = data_train_sample) +\
    geom_density() +\
    labs( x = "place_id", y = "Density", title = "place_id distribution")
```

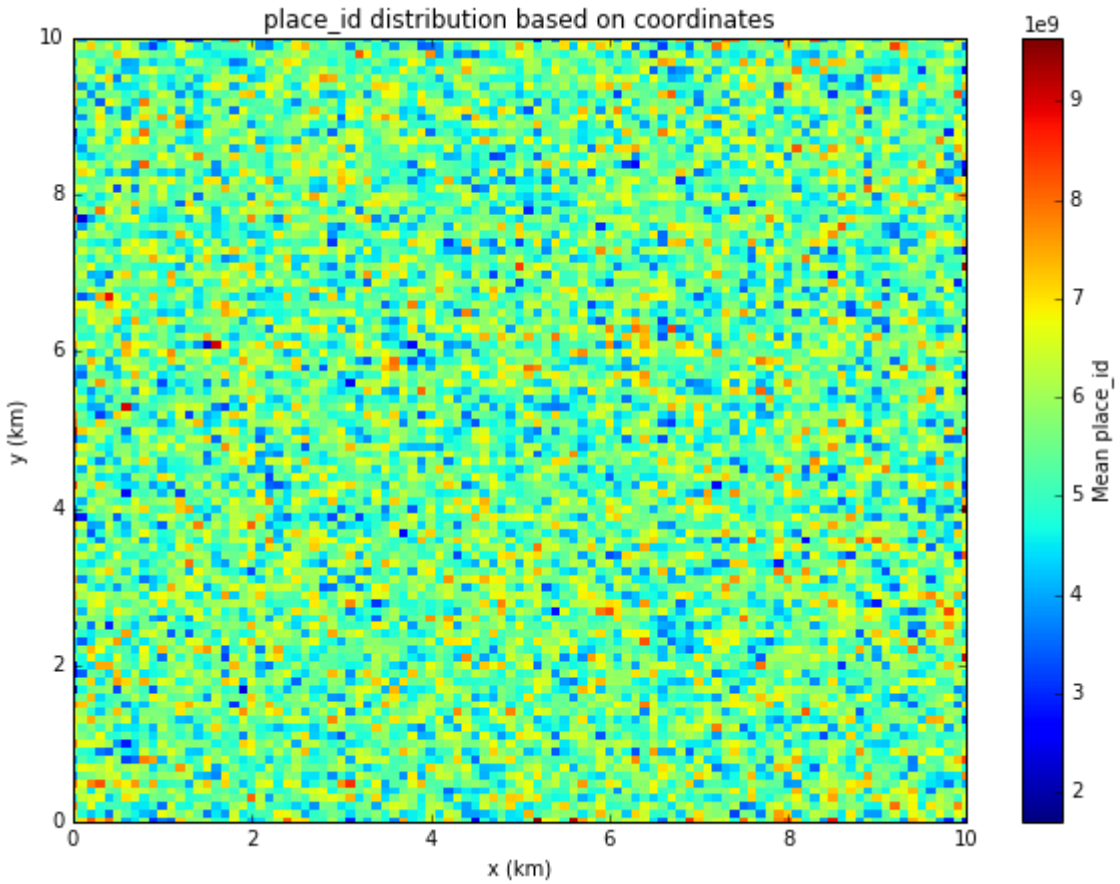


Out[22]: <ggplot: (278027549)>

So place_id is also uniformly distributed, like x and y. We can explore if there is correlation between place_id and coordinates.

In [25]:

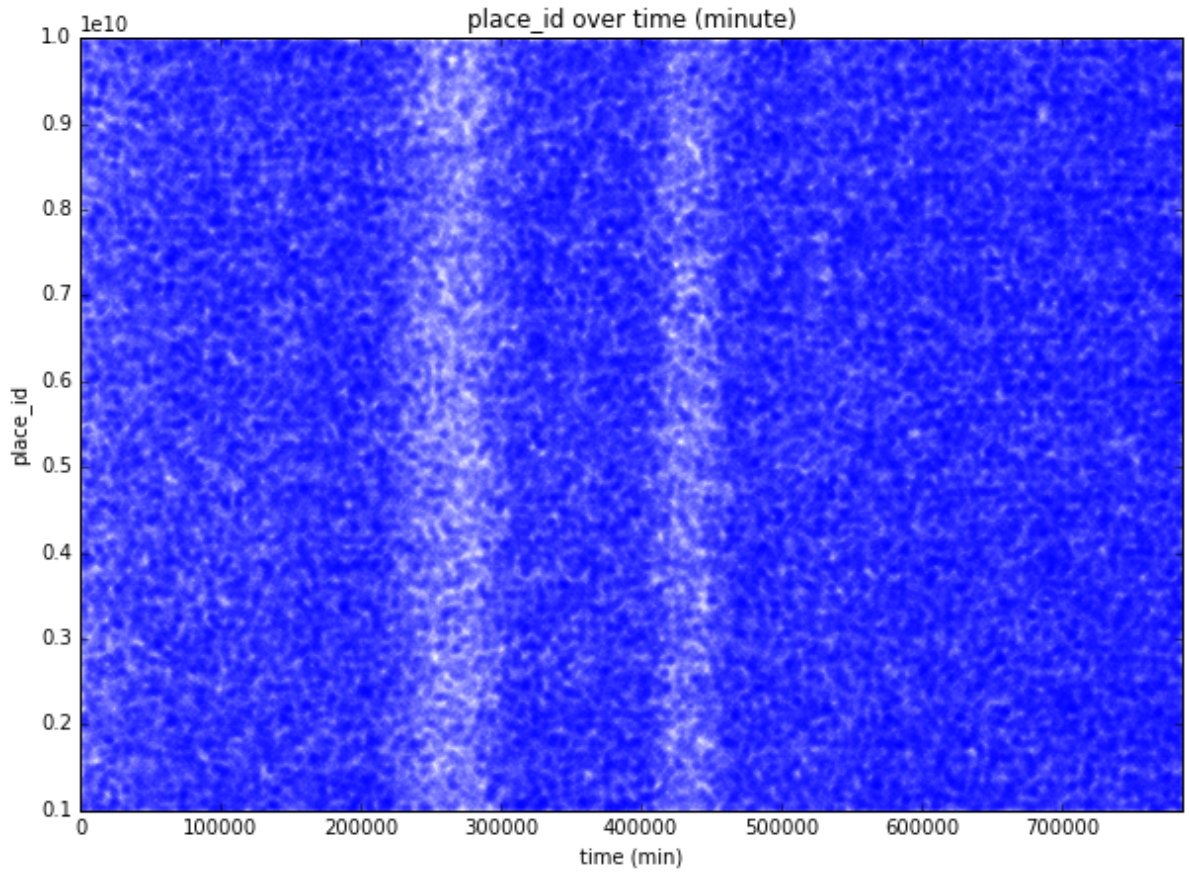
```
data_groupxy = data_train_sample.groupby(["xround", "yround"]).agg({"place

idx = np.asarray(list(data_groupxy.index.values))

plt.figure(figsize=(10,7))
plt.scatter(idx[:,0], idx[:,1], c=data_groupxy["place_id"], marker='s', lw
plt.colorbar().set_label("Mean place_id")
plt.xlabel("x (km)")
plt.ylabel("y (km)")
plt.xlim(0,10)
plt.ylim(0,10)
plt.title("place_id distribution based on coordinates")

plt.show()
```



From the above plot, there is no bovious correlation between place_id and coordinates.

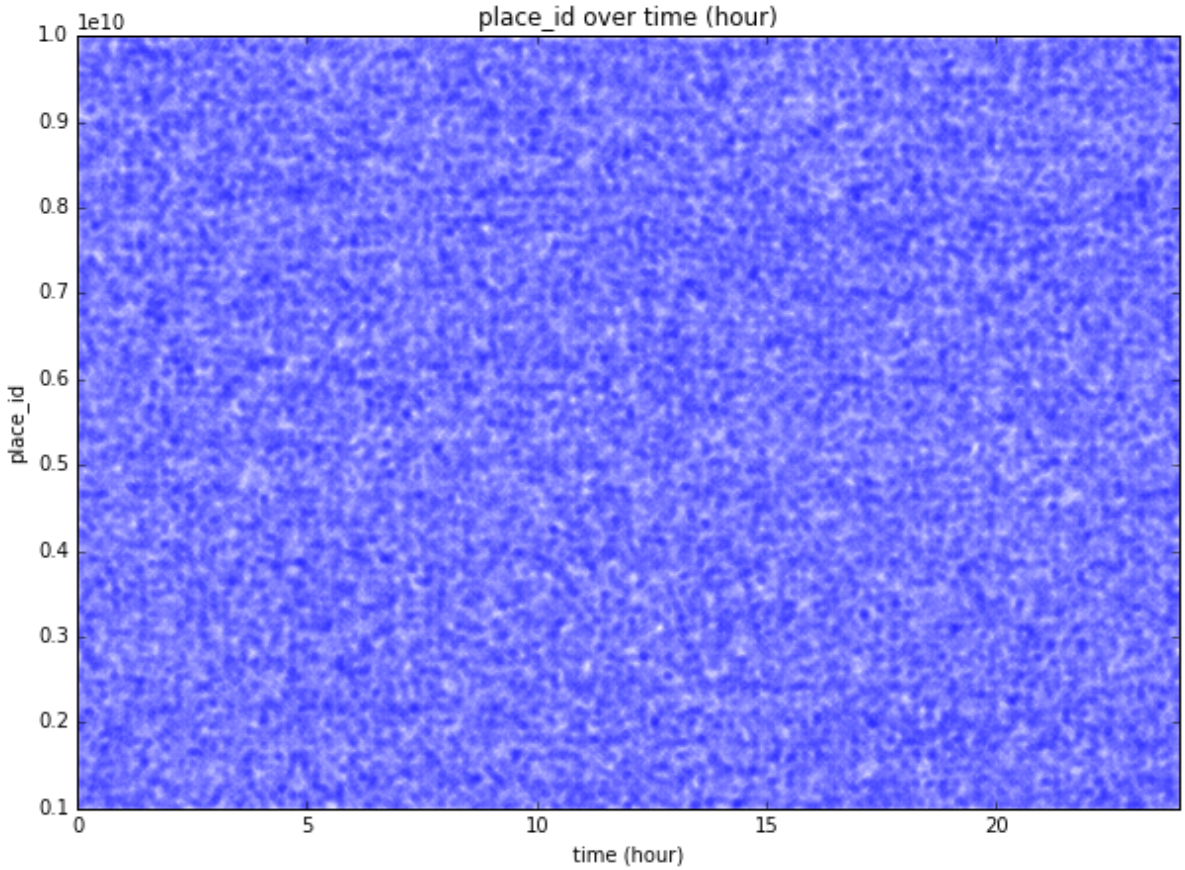
Now let's check if there is correlation between palce_id and time.

In [243]:
```
plt.figure(figsize=(10,7))
plt.scatter(data_train_sample["time"], data_train_sample["place_id"], colo
plt.xlabel("time (min)")
plt.ylabel("place_id")
plt.xlim(0, np.max(data_train_sample["time"]))
plt.ylim(np.min(data_train_sample["place_id"]), np.max(data_train_sample['
plt.title("place_id over time (minute)")

plt.show()
```



We see two white strips, corresponding to the two dips in the distribution of time. That's normal. Other that, we do not observe any thing special. Let's see if the "placie_id" has anything to do with the hour in one day.

In [249]:
```
data_train_sample["time_hour"] = data_train_sample["time"] / 60 % 24
plt.figure(figsize=(10,7))
plt.scatter(data_train_sample["time_hour"], data_train_sample["place_id"],
plt.xlabel("time (hour)")
plt.ylabel("place_id")
plt.xlim(0, np.max(data_train_sample["time_hour"]))
plt.ylim(np.min(data_train_sample["place_id"]), np.max(data_train_sample['
plt.title("place_id over time (hour)")

plt.show()
```



It shows there is no correlation between place_id and the hour in a day either.

Algorithms and Techniques

This is a classification problem and there are some common classifiers which are learned in this course. I will discuss wheather they are proper to be implemented in this problem.

1. Naive Bayes

   Naive Bayes is simple and fast. But it does not fit into a problem with just a few features and a lot of training entries, especiall if the feature values are continuous. This algorithm is not preferred in this project.

1. Support Vector Machine (SVM)

SVM can usually gets high accuracy and can do better generalization. It would be a candidate for this problem, **if** the requirement is to get the most possible place one would like to check into. SVM returns only one output, estimating no other possibilities. So it is not suitable for this perticular problem.

1. Decision Tree and Random Forest

   Decision Tree is simple and fast. Moreover, it can provide multi-outputs, which meets the requirement in this problem. But Decision Tree tends to overfit if the training data set is large. An improvement to it is Random Forest. Random Forest usually gets better generalization. But it also takes more time. I need to find a good balance between efficiency and correctness.

1. K Nearest Neighbor (KNN)

   KNN is simple and can easily do predictions based on the geographic information. Moreover, it can also provide multi-outputs. It fits this problem.

Another challenge is how to find a good way to process such a large volumn of data. I think a practical way is to split the big data set into smaller data sets and process them one by one. To be specific, in this problem, I'm going to split the big city map into smaller grids. In one grid, data set is smaller and there are less classes. I can train one model and test it in a faster way. Of course the size of the grid needs to be determined. I may need to tune the size of the grids to get the best prediction.

Benchmark

This is a tough question. I'm actually not very sure how to prove if my work is good or not. Fortuately, there is a "Leaderboard" in Kaggle for this competition. It lists scores (based on the *MAP@3* metric) of submitters. For now, the highest score is 0.6. I may not be able to get this high. But I think it is practical to get 0.4 or more. So I would like to set the benckmark to be a score of 0.4.

Methodology

Data Preprocessing

In the above exploration, we found the data set is clean and there is no missing value.

There are 4 feature. "x" and "y" is undoutably useful. I will keep them. "accuracy" may be useful. I will keep it. But when the accuracy value is too large, the label is not reliable. So I use accuracy to determine the outliers. "time" is useful. But the original format is not going to be kept. I will transfer the time from minutes to reappearing hours.

I also need to set grids for the data so that I can process smaller data sets and build predictions for the whole map step by step.

First step: get rid of outliers.

The distribution of "accuracy" is not a regular skewed or normal distribution as we see above. It does have a very long tail. What I do in the following cell is to get rid of entries which are considered as outliers at the high end.

In [26]:
```
Q1 = np.percentile(data_train["accuracy"], 25)
Q3 = np.percentile(data_train["accuracy"], 75)

step = 1.5 * (Q3 - Q1)

print "Q3 + step is {}. This value is smaller than the 3rd peak value.".fo
# If I choose the outliers to be entries with accuracy larger than Q3 + st
# the calculated threshold is 147. If we look at the distribution of accur
# find the 3rd peak is larger than this value. So instead of choosing the
# threshold Q3 + step, I chose the 95% percentile to be the threshold.

print "Reselect the threshold to be 95 percentile."
threshold = np.percentile(data_train["accuracy"], 95)

print "Threshold is {}. It's bigger than the 3rd peak value.".format(thres

outliers = []
outliers.extend(data_train[data_train["accuracy"] > threshold].index)

outliers_count = len(outliers)
print "There are {} outliers.".format(outliers_count)

data_train_no_outliers = data_train.drop(data_train.index[outliers]).reset

del outliers
del data_train
```

```
Q3 + step is 147.0. This value is smaller than the 3rd peak value.
Reselect the threshold to be 95 percentile.
Threshold is 208.0. It's bigger than the 3rd peak value.
There are 1448510 outliers.
```

Second step: drop "row_id" and transfer time from increasing minutes to recycling hours.

In [27]:
```
def preprocess(df):
    df['time_hour'] = df['time'] / 60 % 24
    df = df.drop(["row_id", "time"], axis = 1)
    return df
```

In [28]:
```
data_train_good = preprocess(data_train_no_outliers)
```

Third step: setup grids.

In [29]:
```
def set_up_grids(df, m, n):
    x_step = 10.0 / n
    y_step = 10.0 / m
    shift = 0.0001

    pos_x = (df['x'] / (x_step + shift)).astype(np.int)
    pos_y = (df['y'] / (y_step + shift)).astype(np.int)

    if 'grid' in df.columns:
        df = df.drop('grid', axis = 1)

    df['grid'] = pos_y * n + pos_x
    return df
```

In [30]:
```
grid_number = 50

data_train_ready = set_up_grids(data_train_good, grid_number, grid_number)
```

Now let's have a quick look at the good data.

In [31]:
```
data_train_ready.head()
```

Out[31]:

	x	y	accuracy	place_id	time_hour	grid
0	0.7941	9.0809	54	8523065625	21.033333	2253
1	5.9567	4.7968	13	1757726713	13.250000	1179
2	8.3078	7.0407	74	1137537235	1.466667	1791
3	7.3665	2.5165	65	6567393236	7.116667	636
4	4.0961	1.1307	31	7440663949	20.833333	270

In [12]:
```
data_train_ready.describe()
```

Out[12]:

	x	y	accuracy	place_id	time_hour
count	27669511.000000	27669511.000000	27669511.000000	2.766951e+07	27669511.000
mean	4.999899	5.001705	61.938863	5.493896e+09	11.982981
std	2.858127	2.887362	46.055698	2.610821e+09	6.926376
min	0.000000	0.000000	1.000000	1.000016e+09	0.000000
25%	2.534200	2.496500	25.000000	3.223071e+09	5.983333
50%	5.009400	4.988500	61.000000	5.518810e+09	11.966667
75%	7.461900	7.510000	72.000000	7.763973e+09	17.983333
max	10.000000	10.000000	208.000000	9.999932e+09	23.983333

Implementation

Before using the model to get the final predcitions and submit to the online judgement to get the *MAP@3* score. I would like to split the original training data to two data sets for training and **local testing**. I can use F1 score to quickly estimate my model's performance so that I don't need to submit the final predictions and wait long to get the *MAP@3* score. When I think my model is ready, I'm going to submit to get the official score.

In [32]:
```
from sklearn.cross_validation import train_test_split
from sklearn.metrics import f1_score

# function to train and get the f1 score in one square.
def train_test_one_grid(df, n, clf):
    part_data = df[df['grid'] == n]
    features = part_data.drop('place_id', axis = 1)
    labels = part_data['place_id']
    features_train, features_test, labels_train, labels_test = \
            train_test_split(features, labels, test_size=0.25, random_stat

    clf.fit(features_train, labels_train)
    prediction = clf.predict(features_test)

    grid_score = f1_score(labels_test, prediction, average = "weighted")

    return grid_score

# function to train and get the averaged f1 score for selected squares.
def train_test_selected_data(df, clf, grid_selections, func):

    average_score = 0

    for i in grid_selections:
        average_score += func(df, i, clf)

    average_score = average_score / len(grid_selections)

    return average_score
```

Now let's see how what the score is for Decision Tree model. To save time, I select 100 random grids from the total grids and calculate the average score.

In [34]:
```
random_range = random.sample(range(grid_number * grid_number), 100)

from sklearn.tree import DecisionTreeClassifier
clf_DC = DecisionTreeClassifier(random_state=42)
print train_test_selected_data(data_train_ready, clf_DC, random_range, tra
```

```
0.398098580352
```

Now try Random Forest.

In [221]:
```
from sklearn.ensemble import RandomForestClassifier
clf_RF = RandomForestClassifier(n_estimators=10)
print train_test_selected_data(data_train_ready, clf_RF, random_range, tra
```

```
0.447138817201
```

As expected, Random Froest works better than Decision Tree.

Now try K Nearest Neighbor model.

In [151]:
```
from sklearn.neighbors import KNeighborsClassifier
clf_KNN = KNeighborsClassifier(n_neighbors=10, weights = 'distance')
print train_test_selected_data(data_train_ready, clf_KNN, random_range, tr
```

```
0.10738925137
```

Random Forest works the best. But it's too slow. Decison Tree works pretty fast and its F1 score is not much lower than that of the Random Forest. Considering the speed benefit and the potential of improvement, I'm going to use Decision Tree and keep tunning it until it reaches the benchmark.

I also tested the Decision Tree without improvement. I calculated the prediction for the full test data and submitted it to the online judge to see the *MAP@3* score.

In [35]:
```
data_test = pd.read_csv('test.csv')
data_test_good = preprocess(data_test)
del data_test
data_test_ready = set_up_grids(data_test_good, grid_number, grid_number)
```

In [30]:
```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()

# Function to calculate the 3 preditions based on the grid-splitting and t
# The calculation result is saved as a .csv file, which is then submitted
def get_3_predictions(df_train, df_test, clf, grid_selections):

    preds = np.zeros((len(df_test), 3), dtype = int)

    completion = 0.0
    for i in grid_selections:
        df_train_part = df_train[df_train['grid'] == i]
        df_test_part = df_test[df_test['grid'] == i]
        row_ids = pd.Series(df_test_part.index.values)

        features_train = df_train_part.drop('place_id', axis = 1)
        labels_train = df_train_part['place_id']

        le.fit_transform(labels_train)
        clf.fit(features_train, labels_train)
        proba_pred = clf.predict_proba(df_test_part)

        labels_pred = le.inverse_transform(np.argsort(proba_pred, axis = 1

        preds[row_ids] = labels_pred

        completion += 1.0
        print "Completion {}.".format(i)

    df_aux = pd.DataFrame(preds, dtype=str, columns=['l1', 'l2', 'l3'])
    ds_sub = df_aux.l1.str.cat([df_aux.l2, df_aux.l3], sep=' ')
    ds_sub.name = 'place_id'
    ds_sub.to_csv('submission.csv', index=True, header=True, index_label='
    print "Completed!"
```

In [36]:
```
grid_number = 50
whole_range = range(grid_number * grid_number)
```

In [33]:
```
get_3_predictions(data_train_ready, data_test_ready, clf_DC, whole_range)
```

The above line is to use the most simple Decsion Tree model to do predictions for the whole data set. It took about an hour to finish the calculation. After uploading the result to the online judgement, I got a *MAP@3* score of 0.34090, ranking 716/930 at that time. Not very bad for my very first try! I'm more confident that I can reach a *MAP@3* score of 0.4.

I also tried Random Forest model in the following cell:

In [ ]:
```
get_3_predictions(data_train_ready, data_test_ready, clf_RF, whole_range)
```

The above code took around 4 hours to complete, which is 4 times the hours Decision Tree model took. And the *MAP@3* score I got from Random Forest is 0.32461 which is lower than the score of Decision Tree. So I will keep using Decsion Tree as my final model. Another thing we can conclude is that a higher F1 score does not necessarily mean a higher *MAP@3* score for different

algorithms. But it takes much less time to calculate F1 score than to get the official *MAP@3* score. I think F1 score can still give us a hint if the model would work better. In the following section, I will try to tune the model according to the F1 score and *MAP@3* score.

Refinement

At first I want to find which parameters can make the Decision Tree get the highest F1 score. Let's explore parameters for the model.

In [38]:
```
grid_numbers = [40, 50, 80, 100, 200]           # how many grids to split
criterions = ['gini', 'entropy']
min_samples_leafs = [1, 2, 4, 8]
```

In [40]:
```
for grid in grid_numbers:
    whole_range = range(grid * grid)
    random_range = random.sample(whole_range, 100)

    data_train_ready = set_up_grids(data_train_good, grid, grid)

    for i in criterions:
        for j in min_samples_leafs:
            clf_DC = DecisionTreeClassifier(criterion = i, min_samples_lea
            score = train_test_selected_data(data_train_ready, clf_DC, ran
            print "grid: {}, criterion: {}, min_samples_leaf: {}, f1 score
```

The above code gets the following results:

grid	criterion	min_samples_leaf	F1 score
40	gini	1	0.397261838006
40	gini	2	0.397885592696
40	gini	4	0.419931521029
40	gini	8	0.433423771269
40	entropy	1	0.396028793039
40	entropy	2	0.396098585181
40	entropy	4	0.417293722106
40	entropy	8	0.430466947558
50	gini	1	0.400362237227
50	gini	2	0.401271554027
50	gini	4	0.42438916138
50	gini	8	0.437662915603
50	entropy	1	0.400165493042
50	entropy	2	0.400305075361

50	entropy	4	0.422671324099
50	entropy	8	0.433633009142
80	gini	1	0.395078164126
80	gini	2	0.395397765674
80	gini	4	0.417220123764
80	gini	8	0.429069745286
80	entropy	1	0.39370274159
80	entropy	2	0.393712486941
80	entropy	4	0.413489438991
80	entropy	8	0.428474676274
100	gini	1	0.390739659689
100	gini	2	0.391792654317
100	gini	4	0.412581969005
100	gini	8	0.422401300703
100	entropy	1	0.387453724551
100	entropy	2	0.388924549788
100	entropy	4	0.409076673607
100	entropy	8	0.421771627049
200	gini	1	0.372267173075
200	gini	2	0.373894652063
200	gini	4	0.394925276727
200	gini	8	0.401451084958
200	entropy	1	0.375558392119
200	entropy	2	0.375057353041
200	entropy	4	0.392123497231
200	entropy	8	0.399815575129

We can see, 50 grids is the best choice. "gini" works better than "entropy" criterion. And larger min_sample_leaf provides better result.

Now let's increase the min_sample_leaf more to see if the F1 score could be better.

In [45]:
```
grid = 50
whole_range = range(grid * grid)
random_range = random.sample(whole_range, 100)
data_train_ready = set_up_grids(data_train_good, grid, grid)

min_samples_leafs = [8, 10, 12, 15, 20, 25]
for i in min_samples_leafs:
    clf_DC = DecisionTreeClassifier(criterion = 'gini', min_samples_leaf =
    score = train_test_selected_data(data_train_ready, clf_DC, random_rang
    print "grid: {}, criterion: {}, min_samples_leaf: {}, f1 score: {}".fo
```

The above cell gets the following result:

grid	criterion	min_samples_leaf	F1 score
50	gini	8	0.432441945118
50	gini	10	0.433322761725
50	gini	12	0.433543985598
50	gini	15	0.432222033004
50	gini	20	0.428216802144
50	gini	25	0.422653405929

So I'm going to choose min_samples_leaf = 12.

Now, let's see how it performances with the full data set.

In [48]:
```
grids = 50
data_test_ready = set_up_grids(data_test_good, grids, grids)
data_train_ready = set_up_grids(data_train_good, grids, grids)
clf_DC = DecisionTreeClassifier(criterion = 'gini', min_samples_leaf = 12,
whole_range = range(grids * grids)
get_3_predictions(data_train_ready, data_test_ready, clf_DC, whole_range)
```

The above code provides a result of *MAP@3* score = 0.48491. And my ranking at his moment is 626/973. Good, I've reached the bench mark.

I wonder if I can do it better by changing the small squares' width and height.

In [52]:
```
grids_x = [30, 40, 50, 100]
grids_y = [30, 40, 50, 100]

for i in grids_x:
    for j in grids_y:
        whole_range = range(i * j)
        random_range = random.sample(whole_range, 100)
        data_train_ready = set_up_grids(data_train_good, j, i)
        clf_DC = DecisionTreeClassifier(criterion = 'gini', min_samples_le
        score = train_test_selected_data(data_train_ready, clf_DC, random_
        print "grids_x: {}, grids_y: {} criterion: {}, min_samples_leaf: {
            .format(i, j, 'gini', 12, score)
```

This is the result the above cell gets:

grids in x	grids in y	criterion	min_samples_leaf	F1 score
30	30	gini	12	0.434072035171
30	40	gini	12	0.434072035171
30	50	gini	12	0.432649585646
30	100	gini	12	0.435502177202
40	30	gini	12	0.438238698253
40	40	gini	12	0.441216080171
40	50	gini	12	0.434147418199
40	100	gini	12	0.433547337346
50	30	gini	12	0.429828460325
50	40	gini	12	0.43770346585
50	50	gini	12	0.427791076992
50	100	gini	12	0.42737114959
100	30	gini	12	0.434869016113
100	40	gini	12	0.425213953278
100	50	gini	12	0.425467429354
100	100	gini	12	0.414459108481

It seems grid_x = 40 and grid_y = 40 gives me the best F1_score.

In the above, I found 50X50 squares is better than 40X40 squares with min_sample_leaf = 8. Now, I want to compare again which one is better with min_sample_leaf = 12.

In [53]:
```
grids = [40, 50]

for i in grids:
    whole_range = range(i * i)
    random_range = random.sample(whole_range, 100)
    data_train_ready = set_up_grids(data_train_good, i, i)
    clf_DC = DecisionTreeClassifier(criterion = 'gini', min_samples_leaf =
    score = train_test_selected_data(data_train_ready, clf_DC, random_rang
    print "grids_x: {}, grids_y: {} criterion: {}, min_samples_leaf: {}, f
        .format(i, i, 'gini', 12, score)
```

```
grids_x: 40, grids_y: 40 criterion: gini, min_samples_leaf: 12, f1 scor
e: 0.437308982069
grids_x: 50, grids_y: 50 criterion: gini, min_samples_leaf: 12, f1 scor
e: 0.424098480794
```

This is the comparison:

grids in x	grids in y	criterion	min_samples_leaf	F1 score
40	40	gini	12	0.437308982069
50	50	gini	12	0.424098480794

It seems 40X40 model is better.

Now, let's recalculate the prediction for the full data set.

In [58]:
```
grids = 40
data_test_ready = set_up_grids(data_test_good, grids, grids)
data_train_ready = set_up_grids(data_train_good, grids, grids)
clf_DC = DecisionTreeClassifier(criterion = 'gini', min_samples_leaf = 12,
whole_range = range(grids * grids)
get_3_predictions(data_train_ready, data_test_ready, clf_DC, whole_range)
```

The above cell provides a *MAP@3* score of 0.48509, which is better than the previous 50X50 model. So I will keep this one as the final model.

Results

Model Evaluation and Validation

There are several parameters to be determined in the implemention of the prediciton model: the shape and size of the grids, criterion, min_samples_leaf. I used loops to find the best combination of parameters for this Decsion Tree model. What I found is this: when dividing the big city map

into 40X40 smaller squares, using a Decsion Tree with criterion = 'gini' and min_samples_leaf = 12, I can get the best prediction with a *MAP@3* around 0.485 from the testing data set.

The shape and size of the small grids determines how many clusters and training points in a grid. If a Decision Tree needs to work well, it needs to find the best parameters to fit the data set in grids. So in the above tuning process, I need to find the best combination of parameters. And the final result shows that this model works.

I used Decision Tree to predict the outputs. In the above ananlysis, I propose that, for the Decision Tree, as the F1 score increases the *MAP@3* increases. This conclusion is true as the tuning process shows. Because of the nature of this problem, I don't have many ways to prove the robustness of this model. The observation that as F1 increases, *MAP@3* increases provides some proof that the model is robust. Anthor way would be to change the random state of the model and do predictions for the test data to see if the *MAP@3* score is close to what it was. The code is in the following cell:

In [42]:
```
grids = 40
data_test_ready = set_up_grids(data_test_good, grids, grids)
data_train_ready = set_up_grids(data_train_good, grids, grids)
clf_DC = DecisionTreeClassifier(criterion = 'gini', min_samples_leaf = 12,
whole_range = range(grids * grids)
get_3_predictions(data_train_ready, data_test_ready, clf_DC, whole_range)
```

I got a *MAP@3* score of 0.48508, which is very close to the previous value 0.48509. That means our model is robust.

The results found from the model is provided by the official judge so that they are trusted.

Justification

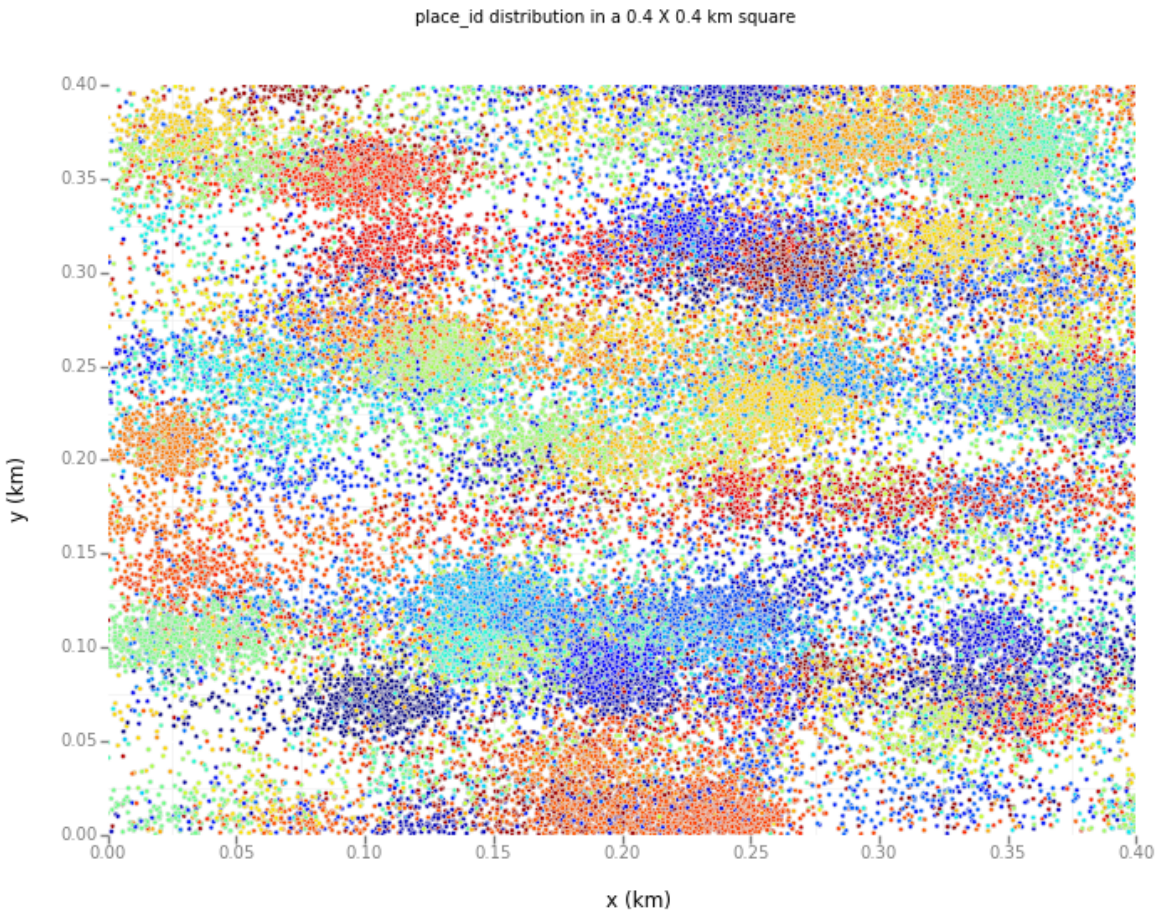
The final result is a *MAP@3* score of 0.485. It is higher than the bench mark 0.4 I set for myself before getting my hands dirty. There is no perticular reasons for choosing this bench mark. It's a competion and I want to get a score as high as possible.

I discussed thoroughly the whole procedure how to get the final model in the above sections. The result is higher than the bench mark and the models is pretty robust.

Conclusion

Free-Form Visualization

The most significant feature of this problem is the size of classes. The data set is big and there are so many "place_id"s in the map. I would like to use the same plot I used above to show how massive the classes are.



Remember, this is just one grid over 625 grids of the whole map. To solve the problem with so many clusters and so large data set, the strategy I used is "divid and conquer". I divided the whole map into smaller grids and trained my model using the data in these grids.

Reflection

The interesting thing I found in this project is that there are so many entries and classes in the data set. The only way for me to solve this problem in a reasonable time is to use the "divide and conquer" strategy. This brings two questions: how to divide the data set and which algorithm to apply. Anthor challenge is that I have to calculate the prediction for the whole big test data set and upload to the online judge to get an estimate of the model performance. That usually takes a lot of time. A better way of estimation should be found.

To improve efficiency, I used F1 scores as a local performance metric and the online judge as the final metric. As I observed, higher F1 score does not necessarily mean higher online judge score for different algorithms, but inceasing the F1 score for the same algorithm does improve the online score. So in the tuning process of a particular model, I can use F1 score as the metric to improve my model.

I investigated the performance of the three different models: Decision Tree, Random Forest and KNN. Random Forest is too slow, not suitable for this particular problem with such a large data set. KNN provides a much worse prediction than the Decsion Tree does. So I chose Decision Tree as my candidate for the tuning step.

In the tuning step, I focused on two problems: how to divide the city map and the best parameters for the algorithm. I used loops to find the best combination of parameters to solve these problems. The final model works well. The final result is much higher than the origin bench mark I set.

Improvement

We see the size of each cluster is not exactly uniform, so a dynamic min_sample_leaf for each grid should work better than a fixed min_sample_leaf. A potential improvemnent is to tune the min_sample_leaf for each grid. This approach will definately take a lot of time but in principal I think the model will be improved and a higher *MAP@3* score should be achieved.

In [ ]: