

BAY AREA BIKE RECOMMENDATION APP

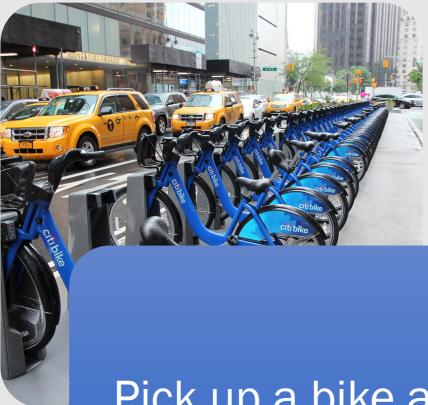
TEAM SPARKLE

ANANT AGARWAL
ARPITA JENA
ASMITA VIKAS
DEENA LIZ JOHN

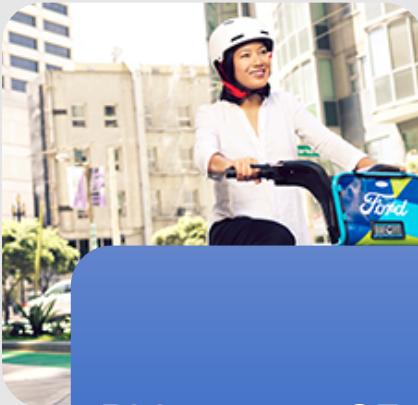


- INTRODUCTION
- RECOMMENDATION APP
- DATA DESCRIPTION
- IMPLEMENTATION IN SPARK
- MODEL BUILDING
- FUTURE SCOPE
- LESSONS LEARNED

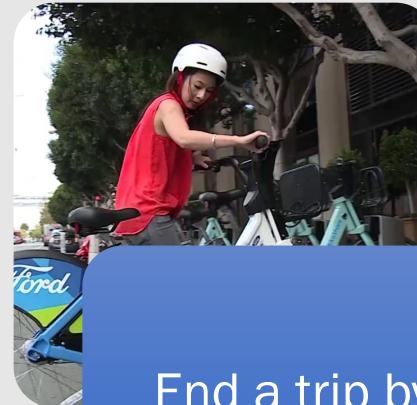
Introduction



Pick up a bike at
any Ford GoBike
station.



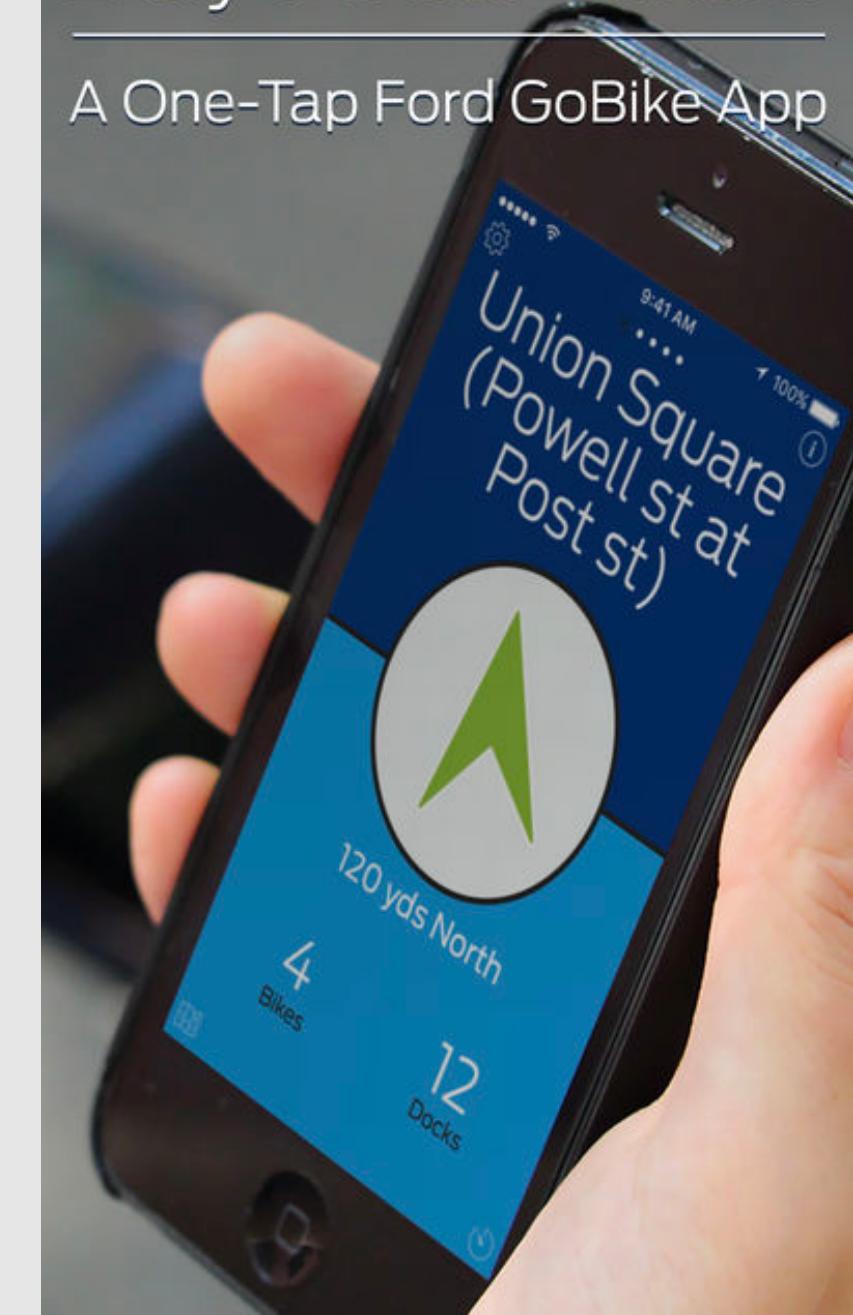
Ride across SF for
quick trips!



End a trip by
returning at any
bike station.

Business Problem

- Given the shortage of bikes, there is a need to optimize availability of bikes to users.
- With the existing Ford GoBike App, real-time bike availability information can be tracked.
- Need a tool to plan future rides.



Recommendation System

- System finds nearest station for future rides based on traffic and bike availability using Google Maps API.
- Users are provided a ranked list of these stations based on predicted bike availability and traffic.



DEMO

GoApp
GoBike Station Recommendation

Location: 101 howard street

Hour: 9

Date: 01-22-2018

Show Reset

Rank	Station_Name	Distance	Duration	Metric
1	Steuart at Market	0.3 mi	6 mins	0.6125
2	Embarcadero at Folsom	0.1 mi	2 mins	0.504166667
3	Beale at Market	0.3 mi	7 mins	0.391447368
4	Spear at Folsom	0.1 mi	3 mins	0.333333333
5	Temporary Transbay Terminal (Howard at Bea	0.1 mi	3 mins	0.165942029

Data Description

- 2.5 GB data from Ford GoBike data release with trip and station information.
- Data from August 2013 to August 2015.
- Nearly 71 million trips.
- Over 15 features of weather information was used as external data.

Pipeline



Implementation in Spark

Data loaded in S3

The screenshot shows the AWS S3 console interface. At the top, there's a navigation bar with the AWS logo, 'Services' dropdown, 'Resource Groups' dropdown, and user information ('Deena Liz John', 'Global', 'Support'). Below the navigation bar, the path 'Amazon S3 > my-msan694' is displayed. The main area has tabs for 'Overview', 'Properties', 'Permissions' (which is selected and highlighted in yellow), and 'Management'. A search bar at the top of the list table contains the placeholder text 'Type a prefix and press Enter to search. Press ESC to clear.' Below the search bar are three buttons: 'Upload', '+ Create folder', and 'More'. To the right of these buttons is the region 'US West (Oregon)' and a refresh icon. The main content area displays a table of five files. The columns are 'Name', 'Last modified', 'Size', and 'Storage class'. The files listed are:

Name	Last modified	Size	Storage class
README.md	Jan 9, 2018 11:13:54 AM GMT-0800	3.7 KB	Standard
station.csv	Jan 12, 2018 4:56:54 PM GMT-0800	5.5 KB	Standard
status.csv	Jan 12, 2018 4:56:54 PM GMT-0800	1.9 GB	Standard
trip.csv	Jan 12, 2018 4:59:40 PM GMT-0800	76.5 MB	Standard
weather.csv	Jan 12, 2018 4:59:50 PM GMT-0800	427.8 KB	Standard

At the bottom of the table, there are two small text boxes: 'Viewing 1 to 5' on the left and 'Viewing 1 to 5' on the right.

EC2 Instance

The screenshot shows the AWS EC2 Instances console interface. At the top, there's a search bar with the placeholder text 'Filter by tags and attributes or search by keyword' and a help icon. Below the search bar is a navigation bar with icons for back, forward, and search. The main content area has a table with the following columns: 'Name', 'Instance ID', 'Instance Type', 'Availability Zone', 'Instance State', 'Status Checks', 'Alarm Status', 'Public DNS (IPv4)', and 'IPv4 Public IP'. The table shows one instance:

Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS (IPv4)	IPv4 Public IP
	i-0d82ea818d7f9aef2	t2.xlarge	us-west-2a	running	2/2 checks ...	None	ec2-54-202-89-240.us...	54.202.89.240

Querying on EC2

The screenshot shows a Jupyter Notebook interface titled "Predictions" running on an EC2 instance. The browser tab is "bay-area-bike-share/" and the notebook title is "Predictions". The notebook contains the following code:

```
In [1]: from pyspark.sql import Row, SQLContext
from pyspark.sql.types import *
from pyspark.sql.functions import *
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml.evaluation import MulticlassClassificationEvaluator, RegressionEvaluator
from pyspark.ml.feature import StringIndexer

In [2]: statusDF = spark.read.format("com.mongodb.spark.sql.DefaultSource").option("uri", "mongodb://127.0.0.1/msan697.status")
stationDF = spark.read.format("com.mongodb.spark.sql.DefaultSource").option("uri", "mongodb://127.0.0.1/msan697.station")
weatherDF = spark.read.format("com.mongodb.spark.sql.DefaultSource").option("uri", "mongodb://127.0.0.1/msan697.weather")
tripDF = spark.read.format("com.mongodb.spark.sql.DefaultSource").option("uri", "mongodb://127.0.0.1/msan697.trip").load()

In [89]: stationDF.printSchema()
root
|-- _id: struct (nullable = true)
|   |-- oid: string (nullable = true)
|-- city: string (nullable = true)
|-- dock_count: integer (nullable = true)
|-- id: integer (nullable = true)
|-- installation_date: string (nullable = true)
|-- lat: double (nullable = true)
|-- long: double (nullable = true)
|-- name: string (nullable = true)

Status Table
In [25]: statusDF = statusDF.withColumn('dayofweek', date_format(from_unixtime(unix_timestamp(statusDF["time"])[0:10], 'yyyy/MM/dd')))
```

Implementing Generalized Linear Regression model to predict Traffic

The screenshot shows a Jupyter Notebook interface with the title "Predictions" in the tab bar. The notebook is titled "Generalized Linear Regression for Traffic".

```
In [110]: from pyspark.ml.regression import GeneralizedLinearRegression  
glr = GeneralizedLinearRegression(family="gaussian", link="identity", maxIter=10, regParam=0.3)  
# Fit the model  
glrmodel = glr.fit(lpoints)  
  
In [111]: # Fit on validation set  
validpredicts = glrmodel.transform(lpoints)  
validpredicts.show()
```

features	label	prediction
[0.0,1.0,4.0,2014...]	-0.5	-0.13092343534083228
[0.0,1.0,4.0,2014...]	-0.6666666666666667	-0.49274437642551777
[0.0,1.0,4.0,2014...]	0.6666666666666667	0.5573660449098027
[0.0,1.0,4.0,2014...]	1.0	0.9459357108710079
[0.0,1.0,4.0,2014...]	1.0	0.95467990402772
[0.0,1.0,4.0,2014...]	1.0	0.9561703654777277
[0.0,1.0,4.0,2014...]	1.0	0.9075470802903581
[0.0,1.0,4.0,2014...]	-0.5	-0.32919074983942453
[0.0,1.0,4.0,2014...]	0.0	0.05532153668088524
[0.0,1.0,4.0,2014...]	0.0	0.06662025170452501
[0.0,1.0,4.0,2014...]	1.25	0.9371638551499849
[0.0,1.0,4.0,2014...]	-1.0	-0.7016509497861689
[0.0,1.0,4.0,2014...]	0.0	0.159911003054665
[0.0,1.0,4.0,2014...]	1.0	0.9549597386306422
[0.0,1.0,4.0,2014...]	0.5	0.46939399384318736
[0.0,1.0,4.0,2014...]	1.6666666666666667	1.3413788312581842
[0.0,1.0,4.0,2014...]	2.9166666666666667	2.197973250460753
[0.0,1.0,4.0,2014...]	1.0	0.8423845486054035
[0.0,1.0,4.0,2014...]	0.0	0.05451604023050238
[0.0,1.0,4.0,2014...]	0.75	0.5843327402625972

only showing top 20 rows

Implementing Random Forest model to predict Availability

The screenshot shows a Jupyter Notebook interface titled "Predictions" (autosaved). The browser tab is "bay-area-bike-share/ Predictions". The notebook content is as follows:

Random Forest for availability

```
In [113]: rf = RandomForestRegressor(maxBins = 70)
rfmodel = rf.fit(lpoints_availability)

In [114]: rfpredicts = rfmodel.transform(lpoints_availability)

In [115]: # Select example rows to display.
availability_features = rfpredicts.select("features").collect()
availability_prediction = rfpredicts.select("label").collect()

In [63]: # Select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(
    labelCol="label", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(rfpredicts)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
```

Root Mean Squared Error (RMSE) on test data = 2.43219

Future Scope

- Adding external datasets like SF crime data and topology data to improve model.
- Integrate real-time bike availability to update suggested future rides.
- Track location of user to suggest new stations.
- Recommend stations for bike drop-offs based on dock availability before even starting a trip.

Lessons Learned

- Cluster capacity and CPU memory should be in proportion with the volume of data.
- Caching the final tables after processing helped in quick modeling, avoiding reruns of the entire code.
- Writing SQL queries for feature engineering was easier by registering dataframes in the table catalog.
- Correct merge conflicts for Git repositories!

THANK YOU