Developing a Bike Sharing Recommender System for Divvy

Calvin Wong[1] , Greg Embree[2] , and Harrison Kiang[3]

Abstract— **There are a lot of factors that can affect ones commute: traffic, weather, seasonality, etc. This is true for any medium of transportation, whether it be driving, biking, walking, taking the bus or subway. We examined how large of a factor each of these play in affecting commute times, commute activity, etc, when applied to the realm of bike-sharing.**

We developed a recommender system that allows users to figure out the best station of origin as well as the best destination station to use for a Divvy trip. Our recommender was able to achieve a 40% top-5 and 50% top-10 accuracy rate.

I. BACKGROUND

Bike sharing applications are simplistic in the information they provide. As it stands, Divvy riders may only have access to information regarding where bikes may be rented and how many bikes are available to be rented at that location. While this is helpful, there are many other factors that go into selecting where to take a ride from. Weather, general traffic, seasonality,and major local events are all factors that could affect commute times and commute activity. Given information ahead of time, the rider may be able to make more informed decisions about the logistics of their trip. For example, if a particular intersection is known for many red light camera violations a rider might choose a different route to take so as not to put them in harms way why while riding. Further, if a major music festival occurring close to the destination of a proposed trip, the rider might seek out an alternative Divvy station from which to end their trip so as not to run into the problem of not having a dock to return the bike. There are plenty of datasets available to provide historical and live information corresponding to these factors. We can use this data to provide better insight as to when the optimal conditions for choosing a Divvy station or intended route of travel might be.

II. INTRODUCTION

Bike sharing programs in many cities, often dominated by a single company, have made numerous datasets available to the public. They typically provide important data points collected since the program's existence. Some of these companies include Divvy in Chicago, Capital Bikeshare in Washington, D.C., CitiBike in New York City, Pronto in Seattle, BIXI Montreal in Montreal, etc. (see Competitions below). Divvy alone supports between 3-4 million rides annually. With the growing popularity and importance of bike sharing it is likely that more effort will be put forth to improving the efficacy of these systems.

A recommender system seeks to predict a particular preference or rating that a user would give to an item. They have extensive applications in search queries, media, products just to name a few. There are two major implementations of recommender systems. First, collaborative filtering is a method that uses large amounts of information from the behavior, preference, or activities of users to predict what users will like based on how similar they are to other users. It is based on the premise that because person A and person B agreed in the past they will agree in the future. While the collaborative filtering is effective it is prone to problems such as scalability and sparsity. Content-based filtering is based on the characteristics of an item and what the users preferences are. This type of system will provide recommendations using what a user has done in the past or is doing in real-time.

While past efforts have utilized publicly available, static datasets to predict system-wide demand, there are few efforts that utilize live datasets from multiple sources to recommend destination stations to users to help them make informed decisions as how to best utilize the systems themselves at a ride level. An example of combining multiple datasets to help inform riders at the rider level would involve a combination of data sources such as weather and general traffic data in conjunction with the live JSON feed provided by Divvy to ensure that there is enough capacity in the system to support each resultant recommendation.

It would be interesting to be able to combine both live and static data sources and datasets, respectively, to make an informed recommendation as to what origin and destination station a rider should choose for a given Divvy trip. The bike sharing community as a whole stands to benefit from using such a system. In this paper, we introduce a recommender system for the Divvy Bike Share program that utilizes a combination of trip information from Divvy and weather information from NOAA to help the user make informed decisions as to which destination station to ride to. Our recommender system was able to achieve around a 40% top-5 and 50% top-10 accuracy rate.

III. RELATED WORKS

A. Competitions

- **Bike Sharing Demand - Kaggle**
  https://www.kaggle.com/c/bike-sharing-demand

[1]C. Wong is a student of the Master's of Computer Science, Data Science program at the Department of Computer Science at University of Illinois, 201 N Goodwin Ave, Urbana, Illinois 61801, USA. Wong is also a Software Engineer at Hyatt Hotel.

[2]G. Embree is a student of the Master's of Computer Science, Data Science program at the Department of Computer Science at University of Illinois, 201 N Goodwin Ave, Urbana, Illinois 61801, USA. Embree is also a trader at Twitch, LLC.

[3]H. Kiang is a student of the Master's of Computer Science, Data Science program at the Department of Computer Science at University of Illinois, 201 N Goodwin Ave, Urbana, Illinois 61801, USA. Kiang is also a Software Engineer at Cox Automotive.

In this competition, participants are asked to combine historical usage patterns with weather data in order to forecast bike rental demand in the Capital Bikeshare program in Washington, D.C.

- Bike Share Analysis - Kaggle
  https://www.kaggle.com/samratp/bike-share-analysis/data
  The data consists of BikeShare information for three large cities in the US - New York City, Chicago, and Washington, DC.
- Cycle Share Dataset - Kaggle
  https://www.kaggle.com/pronto/cycle-share-dataset
  There are 3 datasets that provide data on the stations, trips, and weather from 2014-2016. Pronto is based in Seattle.
- Deep Exploration on Sharing Bike Data
  https://www.kaggle.com/microtang/deep-exploration-on-sharing-bike-data/data
  BIXI Montral is a public bicycle sharing system serving Montral, Quebec, Canada. Movement history and condition of stations data provided. Geographic boundaries of Montreal also provided.

B. Papers

- Bicycle-Sharing System Analysis and Trip Prediction:
  https://arxiv.org/pdf/1604.00664.pdf
  Uses historical weather data from 2014 and properties of Divvy stations and subscribers/users to predict trips in Chicago
- Forecasting Bike Sharing Demand:
  https://cseweb.ucsd.edu/classes/wi17/cse258-a/reports/a050.pdf
  Predicts hourly demand of bikes for Washington D.C. neighborhoods using census and historical weather data
- Multi-Agent System for Demand Prediction and Trip Visualization in Bike Sharing System:
  http:////www.mdpi.com/2076-3417/8/1/67
  Utilizes multiple agents to collect information of various sources, e.g., weather, traffic, bike sharing programs, and data prediction model agents to predict and visualize demand in bike sharing systems. An impressive paper!
- Riding through Divvy Data:
  https://blogs.microsoft.com/chicago/2017/05/19/riding-through-divvy-data/
  Discusses a visualization exercise creating a business intelligence report of a few Downtown Divvy Stations.

IV. Implementation, Initial Architecture

Problem statement: I want to go from A to B. Which Divvy stations should I use to go from A to B?

Super simple overview:

1) User makes request with location information for both origin and destination
2) User receives recommended stations

We engineered an application using streaming semantics provided by Apache Flink, a native streaming framework. We programmed this application in Scala using Gradle as our build tool.

The stream listens to a source/connector that communicates the users request to the application. The application then performs transformations based on live and static datasets from Divvy, NOAA, and other sources. The application then produces the recommendations, which will be ultimately communicated back to the user.

Transformations are based on a variety of data including Divvy station data and weather data. Other data sources such as traffic, sporting events, etc., can also be incorporated into the framework, but are not currently incorporated into our recommender system. More advanced techniques using machine learning, etc., e.g., FlinkML, scikit-learn are utilized to allow for more informed recommendations.

We have provided in our report sample requests, a description of our data flow through a locally runnable Apache NiFi instance, and links to our codebase for the reader to be able to reproduce our results.

A. Data Collection

The two primary sources of data included data made available by Divvy (https://www.divvybikes.com/system-data) and the National Oceanic and Atmospheric Administration's (NOAA) National Centers for Environmental Information (https://www.ncdc.noaa.gov/), formerly the National Climatic Data Center (NCDC).

Divvy has made available historical trip data collected on a quarterly basis from Q3 2013 through Q4 2017. The trip data includes information about each trip such as location, duration, and user information. An example schema of the trip data is shown below:

```
trip_id: ID attached to each trip taken
start_time: day and time trip started, in CST
stop_time: day and time trip ended, in CST
bikeid: ID attached to each bike
tripduration: time of trip in seconds
from_station_name: name of station where trip
    originated
to_station_name: name of station where trip
    terminated
from_station_id: ID of station where trip
    originated
to_station_id: ID of station where trip
    terminated
usertype: "Customer" is a rider who purchased
    a 24-Hour Pass; "Subscriber" is a rider
    who purchased an Annual Membership
gender: gender of rider
birthyear: birth year of rider
```

Listing 1: Schema for trip data. This was pulled from the README of the 2017 Q3Q4 data.

Divvy has also made a live JSON feed of their stations available (https://feeds.divvybikes.com/stations/stations.json). The JSON object returned has two elements: executionTime and stationBeanList. As of the executionTime "2018-03-28 19:53:43", there are 570 children of stationBeanList. This means that the feed,

at this point in time, is tracking information on 570 different stations. An example is shown below:

```
id  2
stationName "Buckingham Fountain"
availableDocks  21
totalDocks  27
latitude  41.876393
longitude -87.620328
statusValue "In Service"
statusKey 1
status  "IN_SERVICE"
availableBikes  6
stAddress1  "Buckingham Fountain"
stAddress2  ""
city  "Chicago"
postalCode  "60605"
location  ""
altitude  ""
testStation false
lastCommunicationTime "2018-03-28 19:49:53"
landMark  "15541"
is_renting  true
```

Listing 2: Sample station data pulled from the JSON feed as of executionTime "2018-03-28 19:53:43"

NOAA's National Centers for Environmental Information has made available their Dataset Discovery tool (https://www.ncdc.noaa.gov/cdo-web/datasets). The application used for this project was the Global Hourly Data application(httpas://www.ncei.noaa.gov/access-ui/data-search?datasetId=global-hourly). The low, high, and observed temperatures for stations within the bounding box of Chicago were pulled from June 1, 2013 through December 31, 2017. Note that these dates correspond with the Divvy trip data. The data was downloaded and manually cleaned to a more standard CSV format where the values did not include commas. An example of the pulled data is shown below:

```
station,name,state,country,date,tmax,tmin,
    tobs
USC00111550,CHICAGO NORTHERLY ISLAND,IL,US
    ,2013-06-01,75,59,65
```

Listing 3: Sample Chicago area weather data pulled from the Global Hourly Data application from NOAA's Data Discovery page (https://www.ncdc.noaa.gov/cdo-web/datasets).

Note that the NOAA weather data has multiple data entries for each station and date in the Chicago area. The average of the tmax, tmin, and tobs were taken from all stations such that a single entry was formed to represent the temperature of Chicago at a given day. The logic is that temperature would not vary much from station to station, therefore, it is reasonable to persist the mean, per date, for the entire city.

B. Model Development with FlinkML

In addition to data collection, we also attempted using Flink for model development, in particular Flink Machine Learning (FlinkML) library. But we did not get good recommendation results from it and therefore we switched to other technologies, which we will cover in later section. In this section however we would like to talk about our choice of Flink and what we did with it.

Flink offers two advantages for model development. First of all, Flink excels in both batch and real-time processing. This allows us to build our recommender system to refresh recommendations in batch or in real-time. Secondly, Flink has a machine learning library built-in (FlinkML) and this allows us to try various algorithm with minimal coding.

1. Batch and Real-time in one Implementation. Flink implements Kappa architecture, which focuses on stream processing in real-time. To Flink, a stream is an endless continuous flow of data. On the contrary, a batch is a stream with a beginning and an end. Basically Flink does stream processing at the core but the difference between batch and real-time processing is how the stream of data is fed into processing. As a result, we can provide one implementation of recommendation logic to use with Flinks DataStream API for real-time recommendation, and to use Flinks DataSet API for batch recommendation.

2. FlinkML. Flink comes with a machine learning library. At the time of writing, FlinkML in version 1.4 implements ALS (Alternating Least Squares), SVM (Support Vector Machine), KNN (K-Nearest Neighbors Join), SOS (Stochastic Outlier Selection), MLR (Multiple Linear Regression), and etc. In future releases, there would be regularization with LASSO and Ridge, Random Forests, PCA (Principal Components Analysis).

Our first model development was to use ALS in FlinkML. In a typical recommendation use case, there are two features and a label. The two features are usually about the user, and the feature or the precondition of the user. The label would be the prediction or result we are interested in knowing. ALS is perfect for this recommendation use case; the `fit()` function of ALS would train a model using training data with two features and a label. Then the `predict()` function of ALS would take the model and test data with two features; it would then output predicted labels. However, this typical recommendation workflow does not really fit our recommendation use case. For a user checking out a bike from a station, we incorporate the checkout time, weather, events, gender, birth year, and other info to generate a recommended destination station to the user to return the bike to. We did attempt to select a couple feature pairs (for example, origin station ID with checkout hour, or origin station ID with temperature) to train and predict using ALS. The predicted results have R-Squared scores less than 0.5, which is not very good. But to improve predictions, we need to incorporate more than two features. Therefore, we have concluded that ALS is not a good fit.

We then moved onto using MLR (Multiple Linear Regression) in Flink that also provides `fit()` and `predict()` functions. To pass training data and test data into the functions, we needed to first prepare data in DataSet data structure in the type of a LabeledVector. Our features and labels are in CSV format in text. So we first convert CSV data into LibSVM format and then call Flink′s `MLUtils.readLibSVM()` function to get data as La-

beledVector. After running `fit()` to get the model and passing the model and test data into `predict()` however, we found many of the predictions to be non-positive and did not match any of the known station IDs.

The following is a snippet of the results. The last number in the line is the predicted station ID. A lot of them are nonnegative and do not correspond to the station IDs, like a classification problem.

```
(SparseVector((0,11.0), (1,11.0), (2,89.0),
    (3,99.0)),-1.317382434262392E59)
(SparseVector((0,11.0), (1,11.0), (2,31.0),
    (3,49.0)),-6.520742601728 86E58)
(SparseVector((0,11.0), (1,11.0), (2,8.0),
    (3,133.0), (4,1.0), (5,1986.0))
    ,-2.6250065694872936E61)
(SparseVector((0,11.0), (1,11.0), (2,43.0),
    (3,284.0)),-3.6020287621889956E59)
```

Listing 4: Sample output using from FlinkML's SVM.

In other words, none of the predicted label matches with test labels, and using Multiple Linear Regression is not good with our data to provide recommendation through predicting destination stations.

C. *Switching to scikit-learn with Apache Nifi and Kafka*

During the evaluation of FlinkML, we used scikit-learn alongside with the same sets of train/test data and the same classification/clustering methods to generate predictions as benchmark results; we always got a fair amount of good results in which predicted stations matching with test labels. With project's due date rapidly approaching while having little success with FlinkML, we had decided to use scikit-learn as the basis of model development. There were two major changes in making the switch. The first change was programming language. We had been programming in Scala for Flink, but scikit-learn is a python library and therefore our model development had to switch to using python. The second change was execution environment. Our original plan was to use Flink end-to-end, in which Flink would continuously ingest, build model, and make recommendations. Now however, Flink is only responsible for ingestion and we need to architect a separate work flow to run model-building and recommending using scikit-learn, and also this flow has to be able to run for both batch and real-time cases. Fortunately in week 13 of the course, we were introduced Apache NiFi and Apache Kafka that made this work flow possible.

Apache NiFi is a technology that is designed to move and transform data. We have come up with three main NiFi flows:

The first flow is data preparation. As soon as station and trip data was ingested by Flink, Flink would output the file to a folder called ingested; NiFi's ListFolder processor picks it up and passes it onto NiFi's ExecuteStreamCommand processor that runs the parse.py script. The script represents categorical data with numeric values, performs test/train split, and places model files, train files, and test files into model, train, and test folders respectively.

```
def get_random_index_set(size):
    test_size = 0.2 * size
    index = set()
    while (len(index) < test_size):
        index.add(random.randint(0, size-1))
    return index

def get_start_month(start_time):
    month_switch = {
        'Jan': '1',
        'Feb': '2',
        'Mar': '3',
        'Apr': '4',
        'May': '5',
        'Jun': '6',
        'Jul': '7',
        'Aug': '8',
        'Sep': '9',
        'Oct': '10',
        'Nov': '11',
        'Dec': '12'
    }

    s = start_time.split(' ')
    return month_switch.get(s[1], '0')

def get_start_hour(start_time):
    raw = re.sub('^.* ', '', re.sub(':.*', ''
        , start_time))
    formatted = str(int(raw))
    return formatted

def get_gender_num(gender):
    if gender.lower() == 'male':
        return '1'
    elif gender.lower() == 'female':
        return '2'
    else:
        return '0'

def get_birthyear(birthyear):
    if birthyear == '':
        return '0'
    else:
        return birthyear

user_type_switch = {
    'Customer': '1',
    'Subscriber': '2'
}

def get_user_type(user_type):
    return user_type_switch.get(user_type, '0
        ')
```

Listing 5: Code snippet in parse.py to preprocess ingested data.

The second flow is training model. As soon as the train file is available in the train folder, another NiFis ListFolder processor would pick up the train file and pass it onto another NiFis ExecuteStreamCommand processor to run ml.py that outputs models in pickle file to a model folder.

```
def train_and_predict_top_k(clf, x_train,
    x_test, y_train, y_test, k, description):
    y_train = np.array(y_train).reshape(len(
        y_train),)
    clf = clf.fit(x_train, y_train)
```

```
    joblib.dump(clf, description + '.pkl',
        compress=3)
    classes = clf.classes_
    predictions_proba = clf.predict_proba(
        x_test)
    labels = y_test.values.reshape(len(y_test
        ))
    score = accuracy_top_k(predictions_proba,
         labels, classes, k)
    return score
```

Listing 6: Code snippet in ml.py to build model and output model to file.

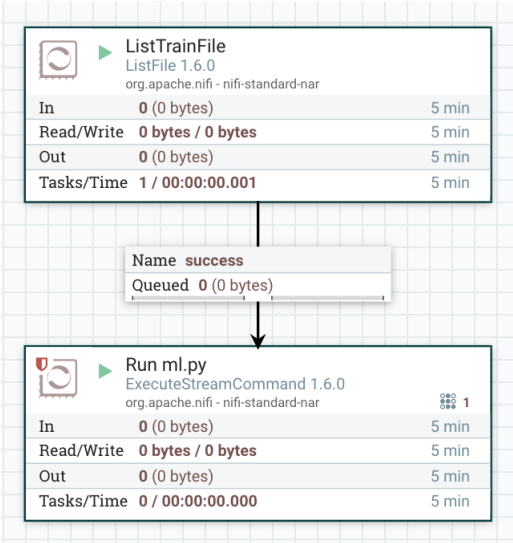

Fig. 2: Second NiFi flow to train model and output to pickle file

The third flow is using model to recommend destination station. This flow has two sources, files for batch use case and Kafka for real-time use case. For batch, as soon as test files or feature-only files are placed in test folder, a separate NiFi's ListFolder processor picks up the file and passes it as FlowFile onto a separate NiFi's ExecuteStreamCommand processor that runs predict.py script. This script loads the model in pickle file and feeds feature-only data into the model to output predictions (i.e recommended destination stations). For real-time, a NiFi's KafkaConsumer processor would consume from a Kafka topic in which each message has one record of features. The processor passes each Kafka message as FlowFile onto the same NiFi's ExecuteStream-Command processor to run predict.py. Since NiFi's FlowFile can vary in size, NiFi made it possible to use the same ExecuteStreamCommand processor to run predict.py for both batch and real-time cases.

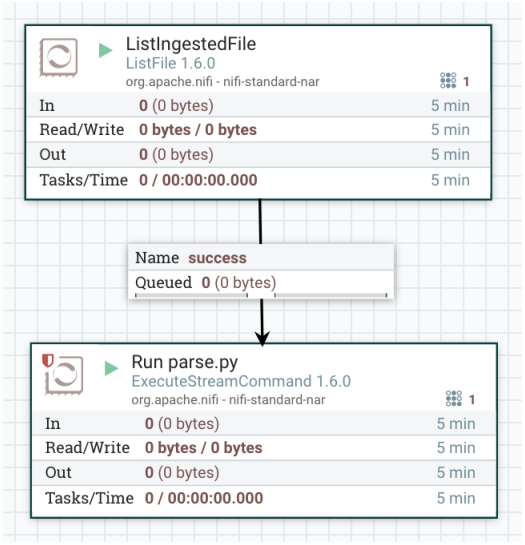

Fig. 1: First NiFi flow to preprocess ingested data

```
clf = joblib.load(model_path)

for line in features.values:
    features = [line[:6]] # clf expects a 2D
        array
    predictions_probs = clf.predict_proba(
        features)
    recommendations = get_top_k_classes(
        predictions_probs, clf.classes_,5)[0]
    recommended_stations = list(map(lambda
        station_id: stations_dict[station_id
        ][1], recommendations))
    recommendations_string = np.array_str(np.
        array(recommended_stations))
    line_str = np.array_str(line)
    sys.stdout.write(line_str+'\t'+
        recommendations_string+'\n')
```

Listing 7: Code snippet in predict.py to load model file and make recommendations.

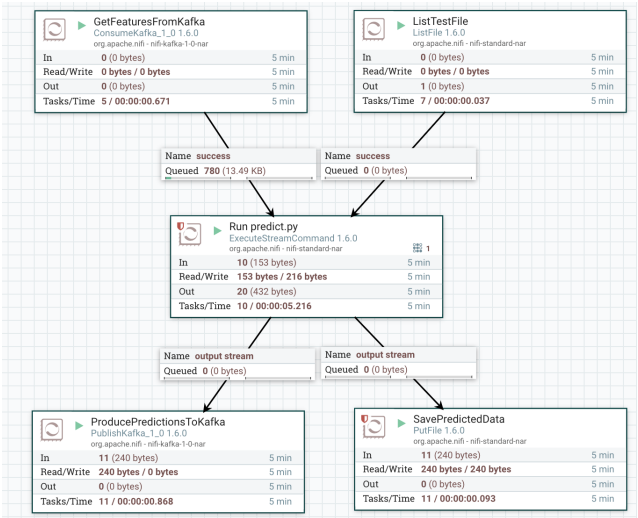

Fig. 3: Third NiFi flow to make recommendations

Using NiFi makes high availability (HA), scaling up, and back-pressure management straight-forward. For HA, NiFi can be installed on multiple servers in cluster mode. Once the flow is deployed in any node in the cluster, it would distribute and run the flow on all nodes. If multiple nodes are down, as long as there is one remaining node, the flow would continue to run. Scaling up the flow is not only made possible by running the flow in cluster, but also by increasing the number

of concurrent tasks at the processor level. This addresses processor hot-spotting that if a particular processor takes longer to process FlowFile than others in the flow, we can increase the number of concurrent tasks for that processor to maintain overall throughput. For back-pressure management, NiFi by default has limited the number of FlowFile in the flow to be 10,000. We can modify this limit according to our cluster size and overall disk space. This ensures there would not be a sudden surge of data going into the flow to overwhelm or even bring down the NiFi flow. NiFi makes all of these possible with just a couple clicks of buttons. It has drastically reduced the amount of development time and allowed us to spent time on model development instead.

V. Flink Data Flow and Integration with Flink

The ingestion process is facilitated by Apache Flink. The Flink ingestion flow is composed of ingestion and transformation jobs that stream data from two primary sources as described in IV-A, namely Divvy and NOAA for their trip and weather data, respectivley.

The machine learning python scripts use scikit-learn to train supervised models on the data generated by the Flink ingestion flow and are plugged into the Apache NiFi dataflow for processing recommendation requests.

A. Flink Ingestion Flow

The Divvy trip data was read in and transformed into a DataSet of `Trip` objects, while the averaged weather data was transformed into a DataSet of `NoaaChicagoDailyTempRaw` objects, which were later transformed into `DateTempMetrics` objects to better represent calendar and numeric data types. The case classes for both `Trip` and `DateTempMetrics` are shown below:

```
case class Trip(trip_id: String, starttime:
    Date, stoptime: Date, bikeid: String,
    tripduration: String, from_station_id:
    String, from_station_name: String,
    to_station_id: String, to_station_name:
    String, usertype: String, gender: String,
     birthyear: String)

case class DateTempMetrics(date: Date,
    avgMaxTemp: Double, avgMinTemp: Double,
    avgObsTemp: Double)
```

Listing 8: `Trip` and `DateTempMetrics` case class definitions.

We used Flink's Dataset API to create `JoinDataSet` of both the trip and weather data. Examples of elements in the resultant `JoinDataSet` are shown below:

```
(Trip(431122,Sat Sep 07 00:00:00 EDT 2013,Sat
     Sep 07 00:07:00 EDT 2013,575,417,31,
    Franklin St & Chicago Ave,54,Ogden Ave &
    Chicago Ave,Customer,,),DateTempMetrics(
    Sat Sep 07 00:00:00 EDT
    2013,86.0,69.0,75.0))
(Trip(431123,Sat Sep 07 00:00:00 EDT 2013,Sat
     Sep 07 00:34:00 EDT 2013,438,2088,81,
    Daley Center Plaza,81,Daley Center Plaza,
    Customer,,),DateTempMetrics(Sat Sep 07
    00:00:00 EDT 2013,86.0,69.0,75.0))
(Trip(431124,Sat Sep 07 00:00:00 EDT 2013,Sat
     Sep 07 00:10:00 EDT 2013,1765,595,19,
    Loomis St & Taylor St,202,Halsted St & 18
    th St,Customer,,),DateTempMetrics(Sat Sep
     07 00:00:00 EDT 2013,86.0,69.0,75.0))
```

Listing 9: A sample of the JoinDataset of Divvy Trips and weather data.

These `JoinedDataSet` objects were transformed into `CCADataPoint` objects to remove duplicate listing of parameters, and to represent the data in a format that can be easily persisted in the form of a CSV file which can be digested by the remaining portions of the data flow, specifically the portions utilising scikit-learn. The schema of `CCADataPoint` is shown below:

```
case class CCADataPoint(trip_id: String,
    starttime: Date, stoptime: Date, bikeid:
    String, tripduration: String,
    from_station_id: String,
    from_station_name: String, to_station_id:
     String, to_station_name: String,
    usertype: String, gender: String,
    birthyear: String, avgMaxTemp: String,
    avgMinTemp: String, avgObsTemp: String)
```

Listing 10: The case class definition of a `CCADataPoint`. This format allows for the persistance of the `JoinedDataSet` of `Trip` and `DateTempMetrics` objects without field duplication.

Note that Apache Flink operates streams in parallel. In the case of a local instance, each thread processes streams independently wherever possible. In this case, since our local machines have 8 threads (4 CPUs with hyperthreading), there are 8 resultant CSV files that are generated when persisting the `CCADataPoint` objects. The files are stored numerically, e.g., `1, 2, 3, 4, 5, 6, 7, 8`, under a directory, e.g., `cca_dataset`. Samples of the CSV outputs within these files are shown below:

```
7564,Sat Jun 29 00:00:00 EDT 2013,Sat Jun 29
    00:16:00 EDT 2013,164,642,76,Lake Shore
    Dr & Monroe St,51,Clark St & Randolph St,
    Customer,,,66.0,62.0,63.0
7565,Sat Jun 29 00:00:00 EDT 2013,Sat Jun 29
    00:13:00 EDT 2013,57,168,33,State St &
    Van Buren St,33,State St & Van Buren St,
    Customer,,,66.0,62.0,63.0
7566,Sat Jun 29 00:00:00 EDT 2013,Sat Jun 29
    00:32:00 EDT 2013,54,1304,44,State St &
    Randolph St,61,Wood St & Milwaukee Ave,
    Customer,,,66.0,62.0,63.0
```

Listing 11: Sample data points persisted to CSV. This is the data ingested by scikit-learn to train and validate supervised models.

B. Model Development with scikit-learn

Our recommendation is engineered to return to the user a configurable number of recommendations as to which destination station is most relevant to the user. Various models were trained and judged on performance using top-10 accuracy. The models that were trainable in a reasonable span of time on our local machines and showed promising results are reported. The training process itself involved

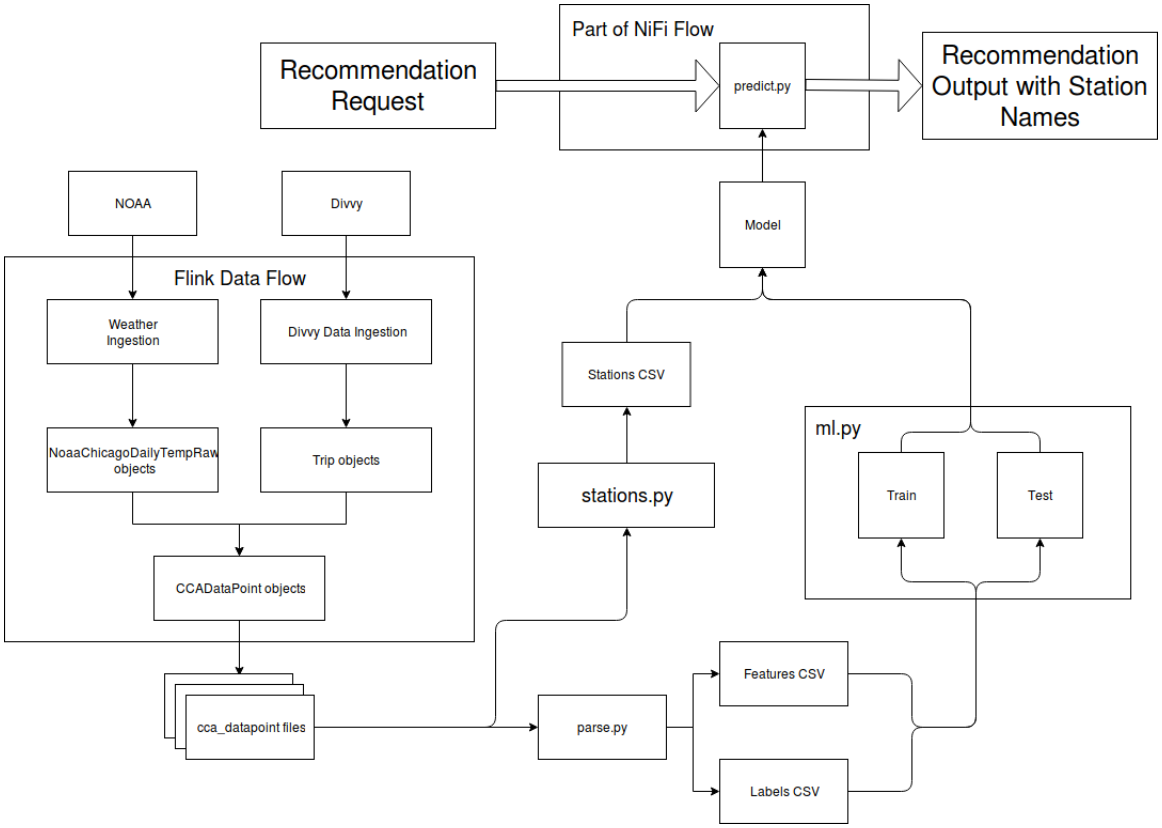

Fig. 4: Architecture diagram. This includes a visualization of the Flink data flow used for ingestion and the scikit-learn scripts used for training and validating supervised models. These same scripts are ultimately plugged into the Apache NiFi data flow for an end-to-end data flow runnable from a local machine.

formatting and splitting the data in preparation for model development, performing the actual training and testing, and finally translating the results back into a meaningful format for the user, namely the station names.

Stations are identified with a `stationName` and `id` as seen in IV-A. The mappings between `stationName` and `id` are generated by processing the `cca_datapoint` CSV files through `stations.py` and stored in `Stations CSV`, as shown in the architecture diagram. These mappings are later used to translate the recommendation results, the `id`s, back to their corresponding `stationName` values for the user.

Relevant features were extracted from the `cca_datapoint` CSV files, namely those features which are presumed to be known before the end of the trip, e.g., the month and hour, the origin station, the gender, birth year, and type of user, and temperature data, and stored in `Features CSV`. A code snippet of the features used is shown in IV-C. The labels consisted of the `id` of stations, and were stored in `Labels CSV`. The lines of both `Features CSV` and `Labels CSV` correspond to one another, i.e., line N of `Features CSV` representing the features of item N correspond to the label stored in line N of `Labels CSV`.

The features and labels stored in `Features CSV` and `Labels CSV` are read by `mp.py` and split into training and testing sets, which are used for model training and testing, respectively. The data splitting and model training were facilitated by `scikit-learn`. Multiple models were trained on datasets that included weather data and didn't include weather data for comparison. The same training and test sets were used for the training and testing of each model. Models trained included Decision Tree classifiers, Random Forest classifiers, and KNN classifiers.

C. Model Selection and Results

Using scikit-learn, we tried using Support Vector Machine K-Nearest Neighbor (KNN), and Decision Tree. The initial results of Decision Tree showed the best results, yielding close to 20% accuracy. As we increased k value in Decision Tree, we yielded close to 50% accuracy, which is the highest we obtained among different clustering/classification methods and configurations.

The Decision Tree classifiers performed the best and were fast enough to train on our local machines. The results of the Decision Tree classifiers for no weather and weather are shown in Figure 5 and Figure 6, respectively. Note that there is nearly a 20% decrease in top-10 performance when combining trip data with weather data. This is surprising, as one would think a richer dataset would yield a more performant model.

The decrease in performance when using weather data could be explained by the process of joining weather data

with trip data. The joins themselves are inner joins, meaning if data couldn't be matched up according to the join clause, in this case when the weather date matched that of the trip date, then the datapoint would be excluded. This is necessary as every data point used for training a model must have valid data for all of its features. This could have led to a decreased number of training datapoints used for the model trained with weather data when compared with all of the trip datapoints able to be collected and used for training the more performant model. This would support the idea that more data could be more important than the quality of data used to train. As always, there are two sides to every coin, and one must observe the advantages and disadvantages of including extra data when training a model.

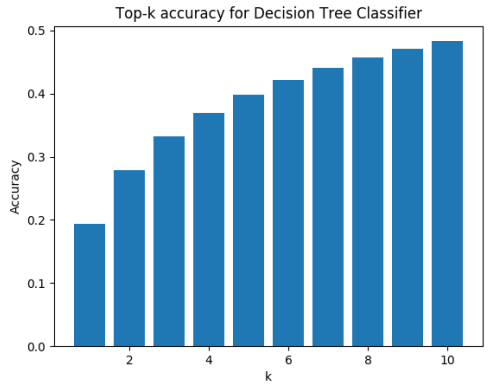

Fig. 5: Bar chart showing k value versus accuracy in decision tree classifier

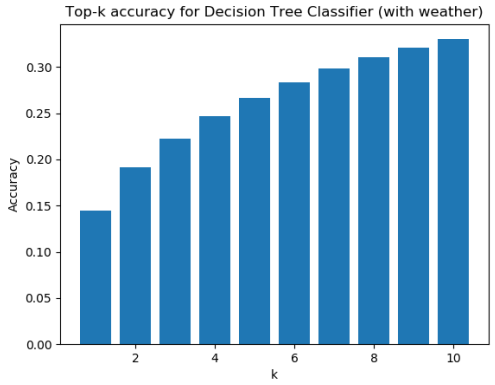

Fig. 6: Bar chart showing k value versus accuracy in decision tree classifier with weather from NOAA. Note that the results are poorer when compared to Fig, 5. This is surprising, as one would think more features would result in a more performant model.

After finalizing model selection of Decision Tree classifier, running the recommender systems end-to-end would yield the following results with features followed by top-five recommended stations :

```
[   9    0  164    1 1986    2   59   83
   68]  ['Damen Ave & Cullerton St' 'McClurg
    Ct & Illinois St' 'Lake Shore Dr & North
    Blvd' 'Canal St & Harrison St' 'Michigan
    Ave & Pearson St']
[   9    0   81    1 1962    2   59   83
   68]  ['Western Ave & Division St' 'Damen
   Ave & Division St' 'Lake Park Ave & 47th
   St' 'Sedgwick St & Huron St' 'Leavitt St
   & Hirsch St']
[   9    0   59    1 1985    2   59   83
   68]  ['Cottage Grove Ave & Oakwood Blvd'
   'Loomis St & Taylor St' 'Lake Park Ave &
   47th St' 'Halsted St & Wrightwood Ave' '
   Ashland Ave & Belle Plaine Ave']
```

Listing 12: Recommender system output

To integrate our recommender system to Divvy technology stack, we would recommend using RESTful API. When user checks out a bike from a station, a GET call is sent to REST endpoints and a JSON response would contain the top-five recommended stations that can display at the station's display or through the Divvy app on the user's phone.

VI. Conclusion

While past efforts have utilized publicly available, static datasets to predict system-wide demand, there are few efforts that utilize live datasets from multiple sources to recommend destination stations to users to help them make informed decisions as how to best utilize the systems themselves at a ride level. An example of combining multiple data

In this paper, we presented a framework for a recommender system of Divvy bike sharing in Chicago based on Divvys system data and weather data from the NOAA. Although our recommendation system does not currently cross reference its recommendations with the Divvy JSON feed to ensure that the system's capacity can support them, we were able to achieve a 40% top-5 and 50% top-10 accuracy rate. Our framework is well structured to integrate this feed to facilitate more reliable predictions, and such efforts can be easily incorporated in future work.

Throughout this process we experimented with a few different tools to use for the development of our model. First, we used the FlinkML libaray to timplement ALS. After seeing low R-Squared scores we realized that it was necessary to incorporate more than two features. Next, we used MLR also of the FlinkML library. However, we experienced a classification problem in that the prediction results resulted in negative station IDs. It was unclear what was causing these negative values.

After little success using FlinkML, we decided to use scikit-learn on the same training and test datasets for our model development. In addition, we made use of Apache NiFi to transform the data flow in our system (Fig 4.). We chose to use SVD, KNN and decision tree classifiers as our models. We witnessed the best results coming from the decision tree which yielded near 50% accuracy. Using these tools we were able to obtain consistent results from our model. As a result, our system can make valid recommendations based on the dataset.

As far as our implementation, there were some alternative routes that could have been taking to make for a more fluid workflow. For example, we considered incorporating the training step in our NiFi flow. Also, the possibility of having Flink output to Kafka. These are things we would consider for improving the design of our workflow.

VII. Future Work

While we are satisfied with our recommendation product there is certainly some future work that can be done. The first thing we would do is integrate our recommendation system so that it is able to cross referencee its recommendations with the Divvy JSON feed to ensure that the system's capacity can support the resultant recommendations. We would also like to do is add a live component that allows for the user to send an HTTP request to the application which would then respond with a station prediction. In addition, it would be very useful if we made our application to be usable in different cities. Another source of further investigation would be to implement datasets that represent more features. Some examples would include examining data on major events, e.g, sporting events, music festivals, and other events that might influence the flow of bikes in the Divvy bike sharing program. We may find that a more robust recommendation results from feature engineering.

APPENDIX

ACKNOWLEDGMENT

We thank the University of Illinois at Urbana Champaign for providing the instruction and guidance for this project as part of the Department of Computer Science's MCS-DS program. We thank the course staff of CS 498: Cloud Computing Applications for the same. We thank Coursera for facilitating the instruction provided through the MCS-DS program. We also thank our respective companies and families for allowing us the time spent on this project.

References

[1] Bike Sharing Demand - Forecast Use of a City Bikeshare System. Bike Sharing Demand, Kaggle, www.kaggle.com/c/bike-sharing-demand.

[2] Samrat. BikeShare Analysis - Analyze Bike Sharing Dataset, Kaggle, Dec. 2017, www.kaggle.com/samratp/bike-share-analysis/data.

[3] Pronto Cycle Share. Cycle Share Dataset - Bicycle Trip Data from Seattle's Cycle Share System. Cycle Share Dataset, Kaggle, 7 Nov. 2016, www.kaggle.com/pronto/cycle-share-dataset.

[4] Tang, Mengran. Deep Exploration on Sharing Bike Data — Kaggle. Deep Exploration on Sharing Bike Data, Kaggle, Nov. 2017, www.kaggle.com/microtang/deep-exploration-on-sharing-bike-data/data.

[5] Zhang, Jiawei, et al. Bicycle-Sharing System Analysis and Trip Prediction. 2016 17th IEEE International Conference on Mobile Data Management (MDM), 2016, doi:10.1109/mdm.2016.35.

[6] Malani, Jayant, et al. Forecasting Bike Sharing Demand. Forecasting Bike Sharing Demand.

[7] Lozano, lvaro, et al. Multi-Agent System for Demand Prediction and Trip Visualization in Bike Sharing Systems. Applied Sciences, vol. 8, no. 1, 2018, p. 67., doi:10.3390/app8010067.

[8] Wei, Kevin. Riding Through Divvy Data. Microsoft Chicago, Microsoft, 19 May 2017, blogs.microsoft.com/chicago/2017/05/19/riding-through-divvy-data/.

[9] Motivate International, Inc, and Divvy Bikes. Divvy System Data. Divvy Bikes, 2018, www.divvybikes.com/system-data.

[10] National Centers for Environmental Information, and NCEI. Climate Data Online: Dataset Discovery. Datasets — Climate Data Online (CDO) — National Climatic Data Center (NCDC), Department of Commerce, 1 Mar. 2018, www.ncdc.noaa.gov/cdo-web/datasets.

[11] Apache Flink Documentation. Apache Flink 1.4 Documentation: Apache Flink Documentation, ci.apache.org/projects/flink/flink-docs-release-1.4/.