Project Report - CS 6240

- Utkarsh Jadhav
- Sriharsha Srinivasa Karthik Kaipa

Running the code:

The source code for the project is located in the folder named "`Finalproject`". The "`src`" folder contains the final code. Make sure that the labelled data for running the project is in the "`input/labelled`" folder and the unlabelled data for prediction is in the "`input/unlabelled`" folder. I have set up "`spark-submit`" to be available directly from the command line. Run instructions for the code are as follows:

- For cleaning the project, building the jar and then running it from the command line, please use the following rules making sure to run them in following order:
  ```
  :~$ make classification-training
  :~$ make classification-prediction
  ```
- For cleaning the project, use the following rule
  ```
  :~$ make clean
  ```
- For building the jar, use the following rule
  ```
  :~$ make jar
  ```

Model and parameter discussion:

The training model chosen for the project is "Random Forest", specifically the RandomForest implementation provided for use in the MLlib Machine Learning library made available for use for running machine learning algorithms on distributed systems using Spark. Random forest training model is an ensemble method for training decision trees on labelled data. Random forests train a set of decision trees separately, so the training can be done in parallel. The algorithm injects randomness into the training process so that each decision tree is a bit different. Combining the predictions from each tree reduces the variance of the predictions, improving the performance on test data. The final prediction by the classifier happens by means of majority voting, where every tree gives a class label as output and the final decision is based on the label which has the highest number of votes.

The parameters to be picked are the number of decision trees in the training model and the maximum depth for each tree. The higher the number of trees, the lower the variance would be, improving the model's test-time accuracy. The maximum depth for the model, in this case, should not be as high as the number of parameters being chosen as features, since, the greater the depth, the more accurate the model is on the training data, which results in overfitting problems. Here is an example of parameters chosen and the accuracy for each set of parameters on the training data that was observed:

```
200 trees 10 depth - 77.45%
150 trees 7 depth  - 76.66%
100 trees 15 depth - out of memory error
```

`100 trees 10 depth - 77.32%`

In the end, 200 trees with a maximum depth of 10 has been chosen as parameters for the model. This choice of parameters comes at the cost of time taken to train the model but gives the highest (albeit by a very very small margin) accuracy.

When deciding on the features to be used for building the model, it is imperative that the features be chosen is such a way that they can be quantifiable and identifiable as a number. It is also a good practice to have as many “useful” features as possible for making the model accurate. But it should also be noted that taking into consideration “all” of the available features will not be helpful as this takes a much longer time for the model to be trained. It is also useful to pick such features that make sense for the labelling being done. For the purpose of this project, the following have been chosen as features for training the model:

- **Latitude**: Decimal latitude.
- **Longitude**: Decimal longitude.
- **Year**
- **Month**
- **Day**
- **Time**
- **POP00_SQMI**: Population per square mile
- **Housing density**: Number of housing units per square mile
- **Housing percent vacant**: Percentage of housing units vacant
- **Elev_GT**: Elevation in meters.
- **Elev_NED**: Elevation in meters. (different resolution)
- **BCR**: Bird conservation region
- **OMERNIK_L3_ECOREGION**: Ecoregions defined by vegetation, animal life, geology, soils, water quality, climate, and human land use.
- **CAUS_TEMP_AVG**: Mean daily average temperature for month in which observation made.
- **CAUS_TEMP_MIN**: Mean daily minimum temperature for month in which observation made.
- **CAUS_TEMP_MAX**: Mean daily maximum temperature for month in which observation made.
- **CAUS_PREC**: Mean total precipitation for month in which observation made.
- **CAUS_SNOW**: Mean snow depth for month in which observation made.

Pseudo code:

The pseudo code for the project is quite simple. The program has 4 explicit map functions that are used for (1) preprocessing the labelled data, (2) preprocessing the unlabelled data, (3) predicting the labels for a part of the training data after the model has been trained and (4) predicting the labels for unlabelled data. There is also an implicit mapreduce job in the code that is used to build the model using the labelled data as training data.

Preprocessing labelled data:

Labelled data that is provided to this program acts as the training data on which a classification model is built. The raw format in which this data is given is not helpful for the program as it is not in the right format for training the model and there is a lot of unnecessary data that is not useful. Hence, the labelled data is preprocessed into the following format:

`<label> <index>:<feature>`

The `<label>` in the above format is the actual label from the labelled data corresponding to an entry. Each of the `<feature>` associated with an `<index>` (which has to begin at 1) is a feature from the labelled data set that is used for training the model. For every record, the entire line is parsed and split based on a "," delimiter since the input data format is in CSV. From the split data entry, only relevant features are picked and given out as processed data. An entry is considered invalid if it's label is anything other than a number, for example "?" or "X". If such a label occurs, the record is discarded. A feature is considered unusable if it's value is anything but a number, for example "?" or "X".

```
labelledPreprocess(line) {
     return <label> <index1>:<feature1> <index2>:<feature2>..
}
```

Here is an example output:

```
0 1:46.6550921 2:-86.1166763 3:2014 4:06 5:161 6:12.12 10:259
12:12 14:50 15:5 16:6 17:6 18:5
```

Preprocessing unlabelled data:

Unlabelled data that is provided to this program is the data for which a label needs to be predicted after training the model. The raw format in which the data is given is not helpful for prediction as it is not in the right format and there is a lot of unnecessary data that is not useful for prediction. Hence, the unlabelled data is preprocessed into the following format:

`<identifier> <index>:<feature>`

The `<identifier>` in the above format is the unique identifier for every record for prediction in the unlabelled data. The original identifier is a string with a leading character of "S" followed by a sequence of numbers identifying the record. For the processed record to be accepted by the model, the leading "S" is removed from the identifier and the remaining numbers are sent instead of `<label>` above. `<index>` and `<feature>` are picked exactly the same as above.

```
unlabelledPreprocess(line) {
     return <identifier> <index1>:<feature1> <index2>:<feature2>..
}
```

Here is an example output:

```
20652129 1:43.2978231 2:-70.5681038 3:2014 4:11 5:327 6:11.67 10:2
11:-.13 12:30 15:3 16:4 17:3 18:7 19:1
```

Classification main:

The main crux of the project lies in this part. The preprocessed labelled data is read into the program as a special format called SVM. The SVM format has two parameters, label and features. Label is a number format and features is a map of indices and features. This format is crucial as the library API call being used to train the model requires its inputs be of this format. After the preprocessed labelled data (70% of it) is used to train the model, the remaining portion is used to test the model built and report on its accuracy. On local testing, the accuracy has been found to be around 77.5% as stated above in this report.

After testing the model on a portion of the training data, the preprocessed unlabelled data is then fed to the model and a prediction for the record along with the record's identifier are output to a single CSV file, after adding a prefix of the character "S" to the identifier.

```
main() {
      // load preprocessed labelled data and split in 7:3 ratio
      model = RandomForest.trainClassifier(trainingData, …)
      testing = testData
            .map(emit(label, model.predict(features)))
      // calculate and print error
      // load preprocessed unlabelled data
      prediction = data
            .map(emit(identifier, model.predict(features)))
}
```

Observation:

While running the program in its current state on EMR, we observed that the number of worker machines did not play a factor in the number of partitions that were produced of the preprocessed labelled data and how many actual machines were utilized by the program. For example, the number of partitions produced for the processed labelled data was observed as 7 regardless whether there were 5 or 10 worker machines. This was happening because of how small the input data is. By simple calculation, it is clear that logically, spark will create only 7 partitions (422 MB into chunks of 64 MB is approximately 6.6 partitions, rounding up to 7 partitions). Hence, there will be a lot of unused machines going idle. To avoid this idleness, number of partitions to create from the labelled data is taken in as argument and is used to partition the data such that all machines are utilized in the preprocessing and model training stages of the program.

The same kind of idle machine behavior is observed while predicting the label for unlabelled data (82.6 MB into chunks of 64 MB is approximately 1.3 partitions, rounding up to 2 partitions). If a similar approach were to be used for prediction as well, the repartitioning of unlabelled data would cause records of unlabelled data to be randomly shuffled around, causing the final output file to not be in the order of the initial data. In accordance with requirement of the project to keep labels in the same order as initial unlabelled data, this

repartitioning is not performed on unlabelled data. This will cause machines to be left idle but it is a price to pay for maintaining order.

Result of repartitioning:

After the above repartitioning has been implemented, it has been observed that there is quite a speedup. Testing has been done on 6 machines (1 master 5 worker) of m4.large capacity. Here is the data observed:

```
Without repartitioning: 1332 seconds run time
With repartitioning: 1074 seconds run time
```

Explanation:

The explanation for the above speedup is because of utilization of all available machines for the purpose of preprocessing the labelled data and for training the model based on the repartitioned preprocessed labelled data.