Basic Approach : Logistic Regression

Logistic regression is used when prediction label has, we had a lot of continuous variables in data. Comprehensive analysis of data suggested us that all the features of our interest could be converted to numeric values. One more reason for selecting LibLinear with logistic regression is that it supports feeding of non-zero values only, hence we reduced the data emitted excessively by just including non zero features. This improved our jobs performance, reduced shuffle and sort cost and gave us a considerably small sized input (230 MBs for Training and 75 MBs for Testing) as compared to approx. 9 gig bz2 input.

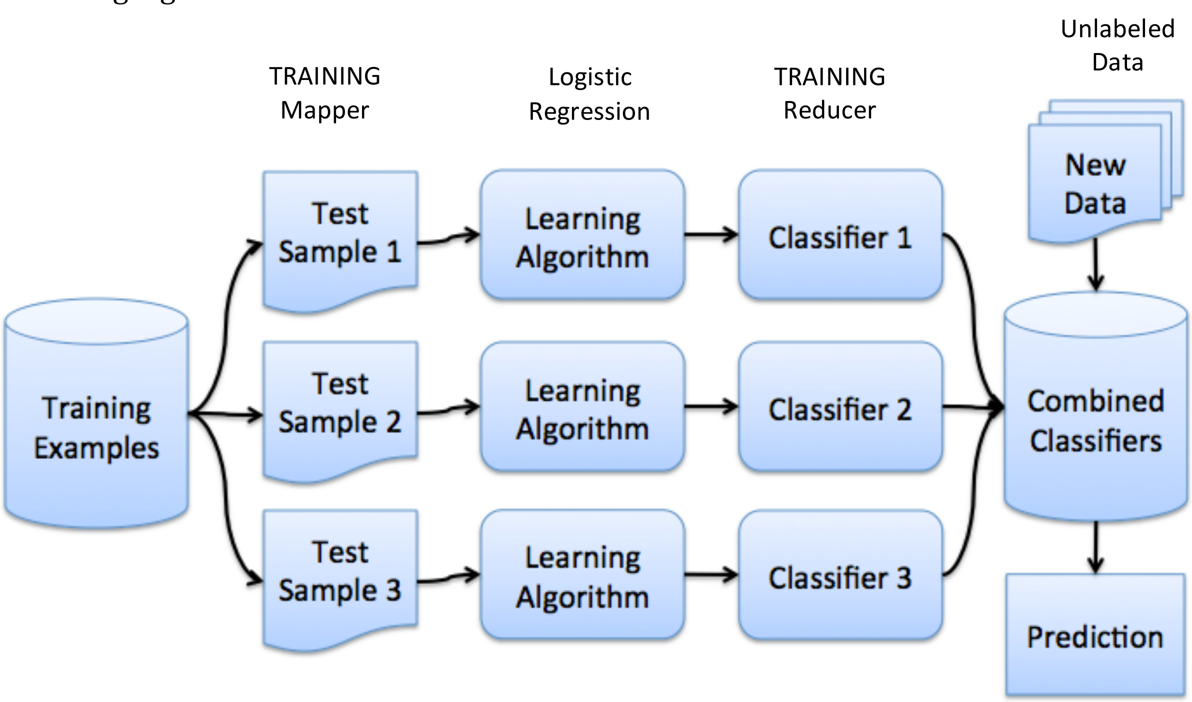
Technologies Used

- MapReduce Platform – Hadoop using Java
- Linear Classification Library – LibLinear
- Model Used : Logistic Regression
- IDE: Eclipse
- AWS: EMR, EC2, S3

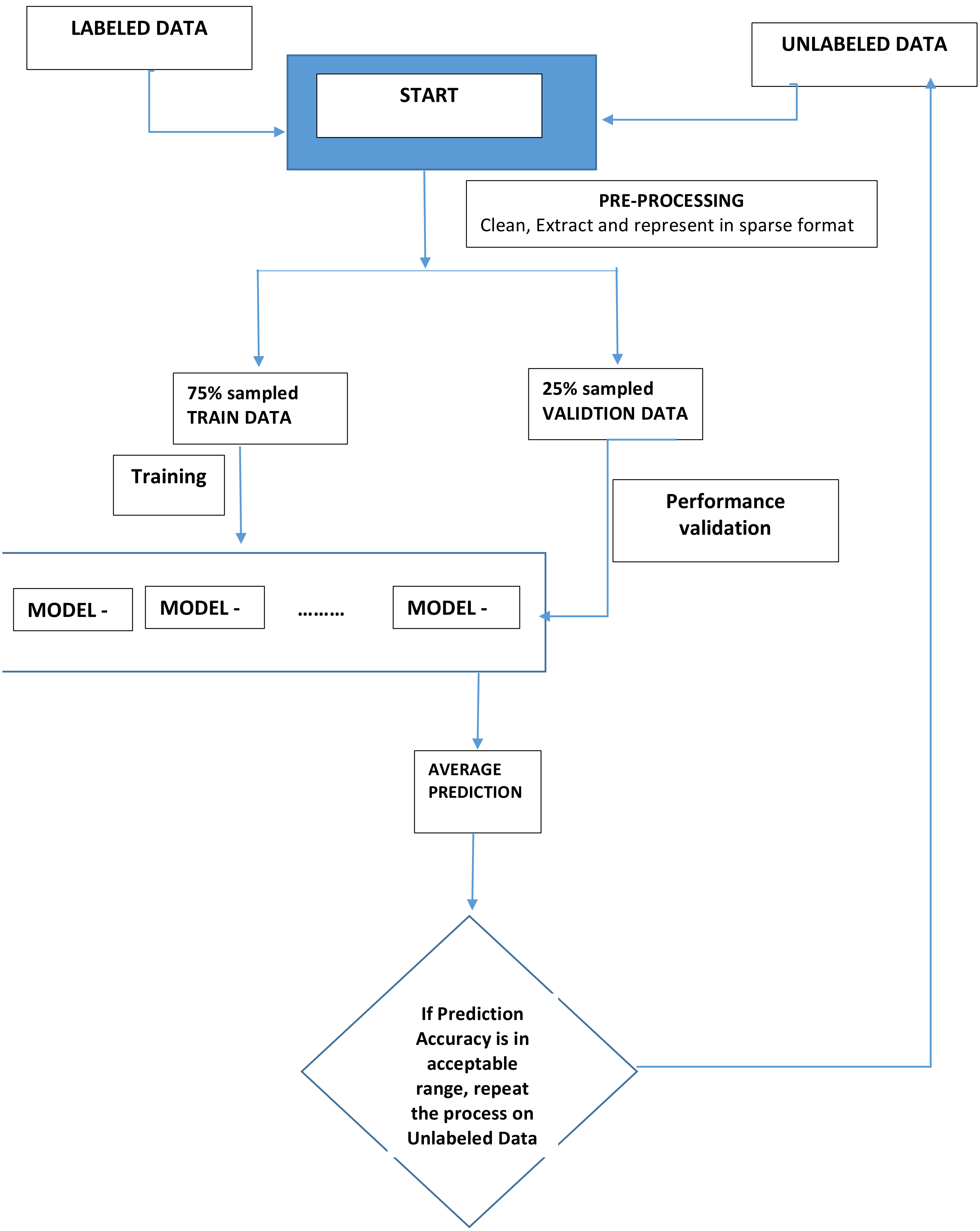
Steps We took :

- Pre-processing – (Cleaning, Parsing, Representation in Sparse Format)
- Training (Bagging and Bootstrapping)
- Prediction (Horizontal Stripes)
- Post-Processing (Parse the prediction result with output expected)

We follow a similar approach with 10 classifiers/models and use logistic regression as a learning algorithm



THE OVERALL PROCESS



Pre-Processing (Data Cleaning):

- We extracted and used important fields i.e **963** columns but represented them in Sparse Matrix format to prevent spilling of data/running out of heap memory
- Implemented very efficient MapReduce Job to extract all non-zero colums from 1000's of columns of interest and randomly sampled them into 75% sample for Training and 25% for testing
- Job uses a single reducer for random sampling

After a considerable thought over support documentation for data and experimentation with data we decided to use all these columns for prediction project

CheckLists	Core	Extended
LocationId	ELEV_NED	DIST_FROM_FLOWING_FRESH
Year	BCR	DIST_IN_FLOWING_FRESH
Month	CAUS_TEMP_AVG	DIST_FROM_STANDING_FRESH
Day	CAUS_PREC	DIST_IN_STANDING_FRESH
Time	CAUS_SNOW	DIST_FROM_WET_VEG_FRESH
Count Type		DIST_IN_WET_VEG_FRESH
Effort Hours		
Effort Distance		
Number of Observers		
All Species		

Total Number of Columns/Features Used : 963

```
// Intention is to clean all the records, scale all values to numeric values since
// we are using logistic regression and only store non-zero values as sparse //matrix to
prevent spilling
Class Mapper{

        map( ..., record t){
                CheklistFeatures = parse(t);
                CoreFeatures  = parse(t);
                ExtendedFeatures = parse(t);

                locationId = CheklistFeatures.getLocationId;

                // Extract all relevant non-zero features from CheklistFeatures,
                //CoreFeatures  & ExtendedFeatures

                for all relevant feature in CheklistFeatures do
                        output = output + (feature.columnNumber, feature)
```

```
        for all relevant feature in CoreFeatures  do
            output = output + (feature.columnNumber, feature)

        for all relevant feature in ExtendedFeatures do
            output = output + (feature.columnNumber, feature)

        emit(locationId,output)
    }

}

// Reducer randomly samples all records to Test and Train output files
class Reducer{
    Random r = new Random();
    Multipleoutpts mos;

    reduce(locationId, [record1, record2….]){

        for all record in [record1, record2….]
            if(r.nextRandom() <= 0.75)
                mos.write("TRAIN", record)
            else
                mos.write("TEST", record)
    }

}

// Send all the records to single reducer since reducer samples this record into split of 75%
for training and 25% for validation
class Partitioner{
    return 0;
}

// Only one reducer is required here since we do a random sampling to divide input split
into 75% for training and 25% for testing
class Driver{
    job.MultipleOutputs("TEST");
    job.MultipleOutputs("TRAIN");
    job.setNumberOfReducerTasks(1);
}
```

Example: Sparse Representation of a record with relevant non zero columns will look like:

0:0,2:2014,3:7,4:211,5:8.0,6:3,7:.25,8:1.609,9:1,105:1,191:1,955:8,956:7,958:9,960:9,962:9

Training : Inspired from DATA MINING II Module

From our training data set D, we created bagged ensemble that is trained as follows:

- Created k = 10 independent bootstrap samples D1, D2,..., Dk of D.
- Train k individual models M1, M2,..., Mk, each separately on a different sample with 10 reducers in the training MR Job

Highlights:

- Every Bootstrap sample is almost of equal size sampled from Dataset uniformly at random, using sampling with replacement
- Each training record has a probability of $1-(1-1/n)^n$ selection, for large n (here 1701975 ~ 1.7 million) this converges to 0.63

Parallel Training Algorithm

```
map( ..., training record r)
    for i=1 to k do
        emit(i, r) with probability p

reduce(i, [r1, r2,...])
    R = load record set into memory
    B = LibLinear.bootstrap(R)
    M = LibLinear.trainModel(B)
    emit(i, M) // Or write to HDFS/S3 file
```

Validation – Map Only Job {Horizontal Stripes}

The bagged model computes the output for a given input X=x as follows:

- Compute Mi(X) for each of the k models M1, Mk.
- Return the average of these individual predictions.

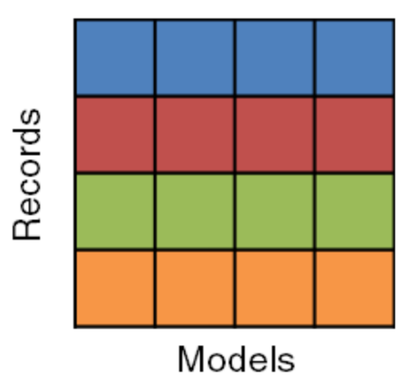
Highlights:

- Each model in the ensemble will calculate its own prediction value for test record, final value is just average of aggregated value over all the models
- We make use of Horizontal Stripe Partitioning for prediction model
- All models are copied via distributed cache in Mapper and average prediction is calculated locally

```
Class Mapper{

    Models = read all models from cache

    map( ..., test record t)
        for each M in Models do
            compute M(t) and update running sum and count
    emit( t, sum/count)
}
```



Prediction & Post-processing

- This is a MapReduce Job that cleans the unlabeled sample data set, it removes all the missing values and extracts all the relevant features.
- Mappers emit SAMPLING_EVENT_ID as key and emits the parsed record to reducer
- Reducer implements Horizontal Partitioning to load models from the distributed cache
- Reducer predicts value for label on record for every model (Here 10 models) and then calculates average of those values
- This average is final prediction and is emitted with key as SAMPLING_EVENT_ID
- A Sequential Job is then run to produce output in same order of sampling id as given in the input

The bagged model computes the output for a given input X=x as follows:

- Compute Mi(X) for each of the k models M1, Mk.
- Return the average of these individual predictions.

```
// Intention is to clean all the records, scale all values to numeric values since
// we are using logistic regression and only store non-zero values as sparse matrix to
//prevent spilling
Class Mapper{

	map( ..., record t){
		CheklistFeatures = parse(t);
		CoreFeatures  = parse(t);
		ExtendedFeatures = parse(t);

		SAMPLING_EVENT_ID = record.getSamplingEventId;

		// Extract all relevant non-zero features from CheklistFeatures,
		//CoreFeatures  & ExtendedFeatures

		for all relevant feature in CheklistFeatures do
			output = output + (feature.columnNumber, feature)

		for all relevant feature in CoreFeatures  do
			output = output + (feature.columnNumber, feature)

		for all relevant feature in ExtendedFeatures do
			output = output + (feature.columnNumber, feature)

		emit(SAMPLING_EVENT_ID,output)
	}
```

```
}


// Reducer randomly samples all records to Test and Train output files
class Reducer{

        Models = read all models from cache
        reduce( …, test record t)
                for each M in Models do
                        compute M(t) and update running sum and count
        emit( t, sum/count)

}


// Sequential Job for maintaining the order in final output file as read in the input with
//prediction values
class OrderSampleIds{

        order(UnLabeledFile data, PredictionOutput p){

                HashMap<String,Integer> samplindIdLookup;
                for every line in p do
                        samplindIdLookup.add(p.samplingId, p.prediction)
                done

                for every line in data do
                        write(line.samplingId ,samplindIdLookup[line.samplingId])
                done
        }
}
```