Online News Popularity Prediction - A Parallelism Approach

EECE5645 : Parallel Processing In Data Analytics

By : Karan Desai, Dhruvi Gajjar & Ishan Palit (Team - 6)

Problem Statement

The dataset summarizes a heterogeneous set of features about articles published by Mashable (an online news organization) in a period of two years. The goal is to predict the number of shares of these news articles. It is a classic regression problem which we will solve using - ***Linear Regression, Lasso Regression, Ridge Regression, Random Forest Regression and Gradient Boosted Trees Regression***. We will use parallel processing at different stages of the implementation of this problem ***(using Spark Mllib)*** and compare the performance with sequential processing ***(using Scikit Learn)***. Further, we will draw inferences based on performance and computation time of these two approaches.

Data Set Link - https://archive.ics.uci.edu/ml/datasets/online+news+popularity

Features	60
Data Points	39644
Target Variable	1

An overview of features -

#	Column	Non-Null Count	Dtype
0	url	39644 non-null	object
1	timedelta	39644 non-null	float64
2	n_tokens_title	39644 non-null	float64
3	n_tokens_content	39644 non-null	float64
4	n_unique_tokens	39644 non-null	float64
5	n_non_stop_words	39644 non-null	float64
6	n_non_stop_unique_tokens	39644 non-null	float64
7	num_hrefs	39644 non-null	float64
8	num_self_hrefs	39644 non-null	float64
9	num_imgs	39644 non-null	float64
10	num_videos	39644 non-null	float64
11	average_token_length	39644 non-null	float64
12	num_keywords	39644 non-null	float64
13	data_channel_is_lifestyle	39644 non-null	float64
14	data_channel_is_entertainment	39644 non-null	float64
15	data_channel_is_bus	39644 non-null	float64
16	data_channel_is_socmed	39644 non-null	float64
17	data_channel_is_tech	39644 non-null	float64
18	data_channel_is_world	39644 non-null	float64
19	kw_min_min	39644 non-null	float64
20	kw_max_min	39644 non-null	float64
21	kw_avg_min	39644 non-null	float64
22	kw_min_max	39644 non-null	float64
23	kw_max_max	39644 non-null	float64
24	kw_avg_max	39644 non-null	float64
25	kw_min_avg	39644 non-null	float64
26	kw_max_avg	39644 non-null	float64
27	kw_avg_avg	39644 non-null	float64
28	self_reference_min_shares	39644 non-null	float64
29	self_reference_max_shares	39644 non-null	float64
30	self_reference_avg_sharess	39644 non-null	float64
31	weekday_is_monday	39644 non-null	float64
32	weekday_is_tuesday	39644 non-null	float64
33	weekday_is_wednesday	39644 non-null	float64
34	weekday_is_thursday	39644 non-null	float64
35	weekday_is_friday	39644 non-null	float64

The target variable is the column **shares** which we are trying to predict.

Data Pre-Processing

On analyzing the data, it is seen that there are no null values and it is already cleaned. Hence, no imputation was needed.

Before any further analysis - we have split the data into train(85%) and test set (15%) randomly sampled from the dataset. All further investigations were on the train set.

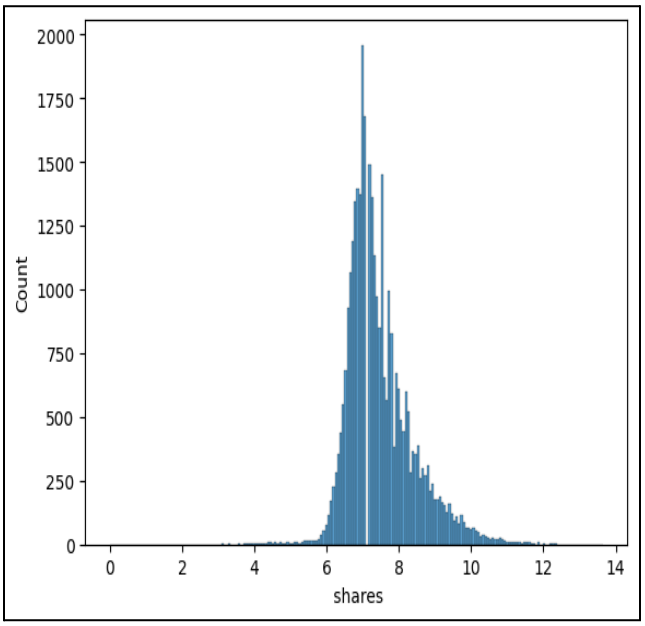
Train	33697
Test	5947

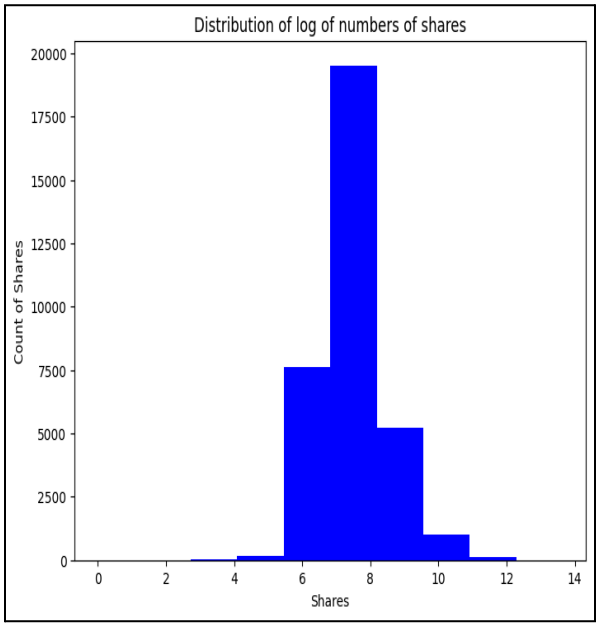
```
# Now we will split data into 2 parts - Test Set (15%) and Training Set (85%) of randomly sampled data before we begin the EDA
from sklearn.model_selection import train_test_split
train, test = train_test_split(raw_df, test_size=0.15, random_state=42, shuffle=True)
```

We have checked the number of features with only 1 unique value and found that none of the features have only a single unique value. This was necessary to remove features which do not add any predictive information to our model.

Two of the features - ***url and timedelta*** are **non-predictive** as stated in the **data dictionary**, hence they were removed. Also, the data is already encoded hence one hot encoding was not needed.

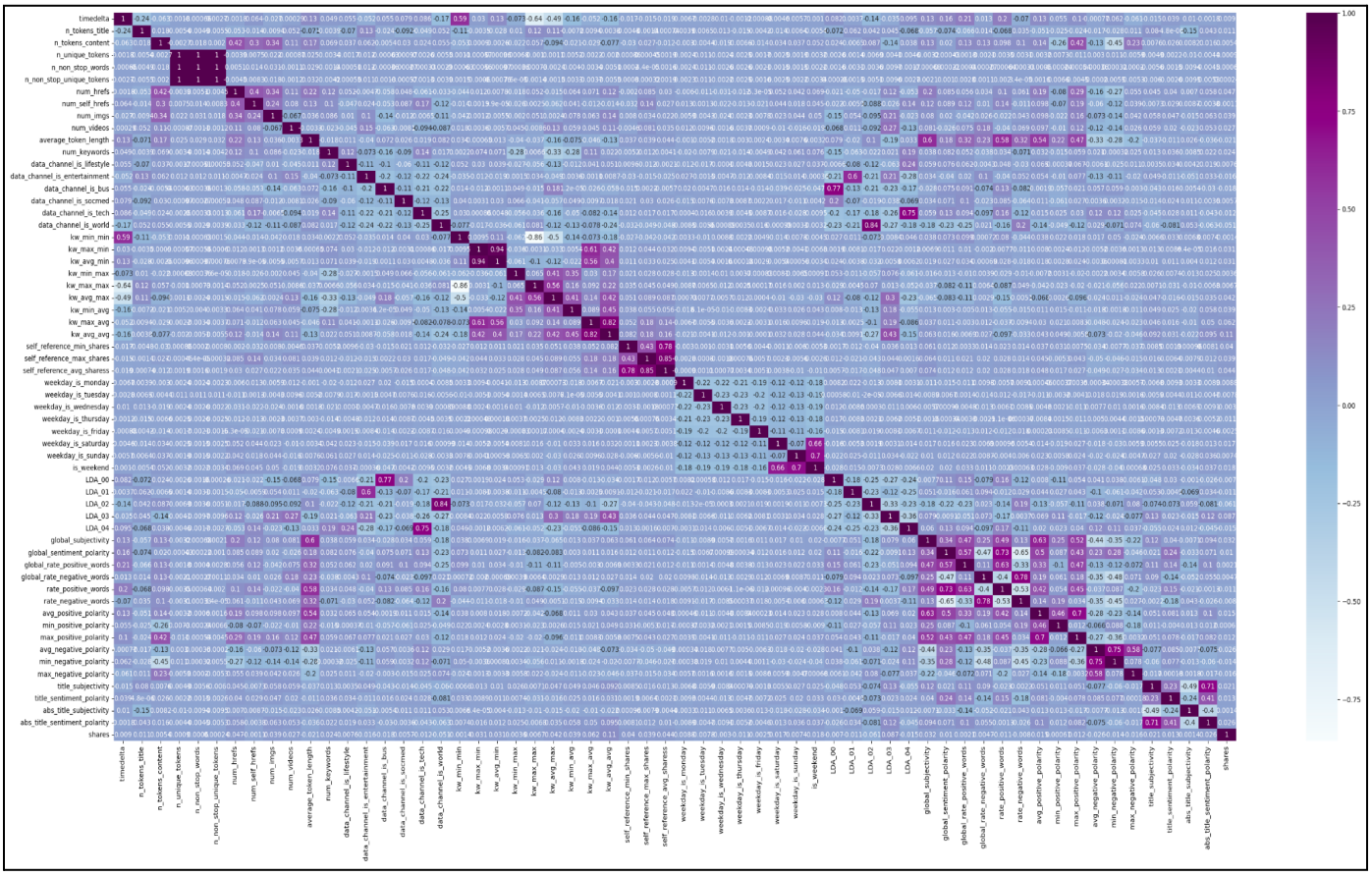
The range of the target variable is very large - so we will perform log transform and then check the distribution.



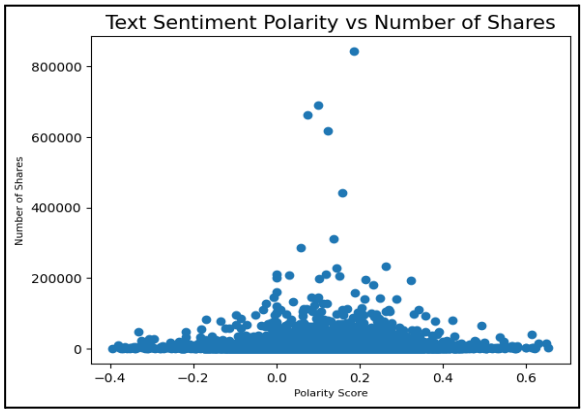


Data Pre-Processing

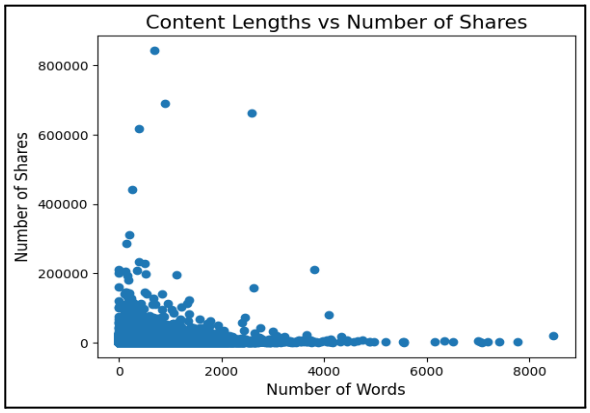
On plotting the feature correlation heatmap we see that the multicollinearity is low among features, so we did not drop any features based on this.



Some other features were also analyzed -

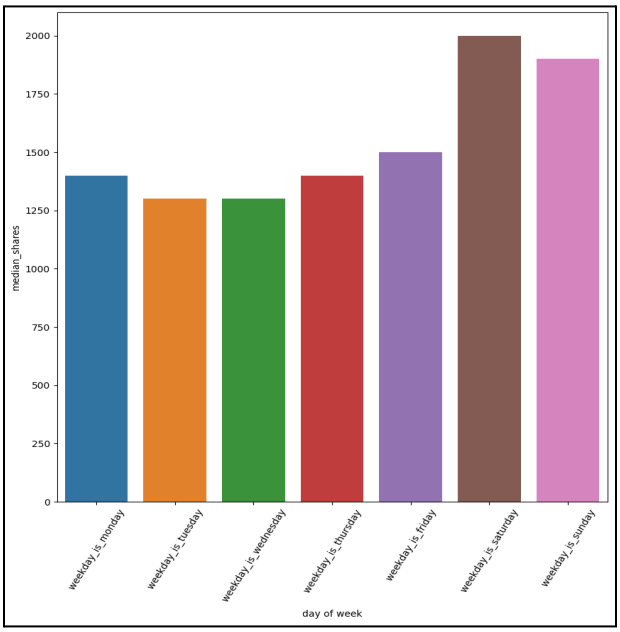


For polarity score is in the range 0.0 - 0.6 we see more shares

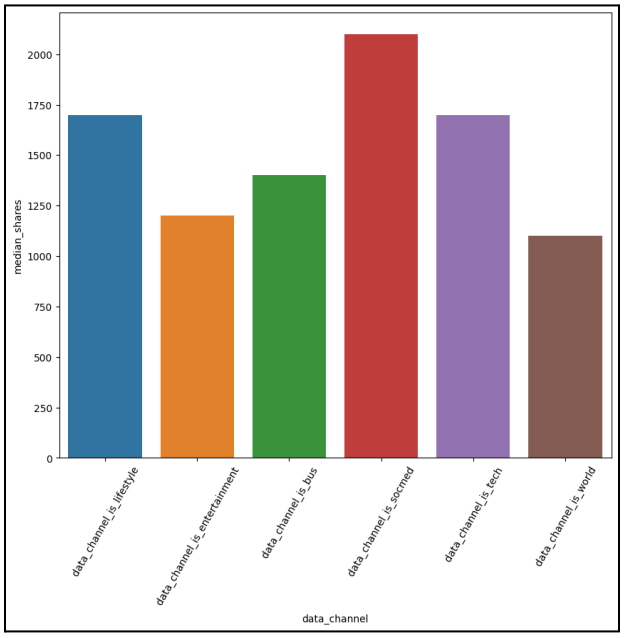


Number of words is in the range 1000-2000 we see more shares

Data Pre-Processing

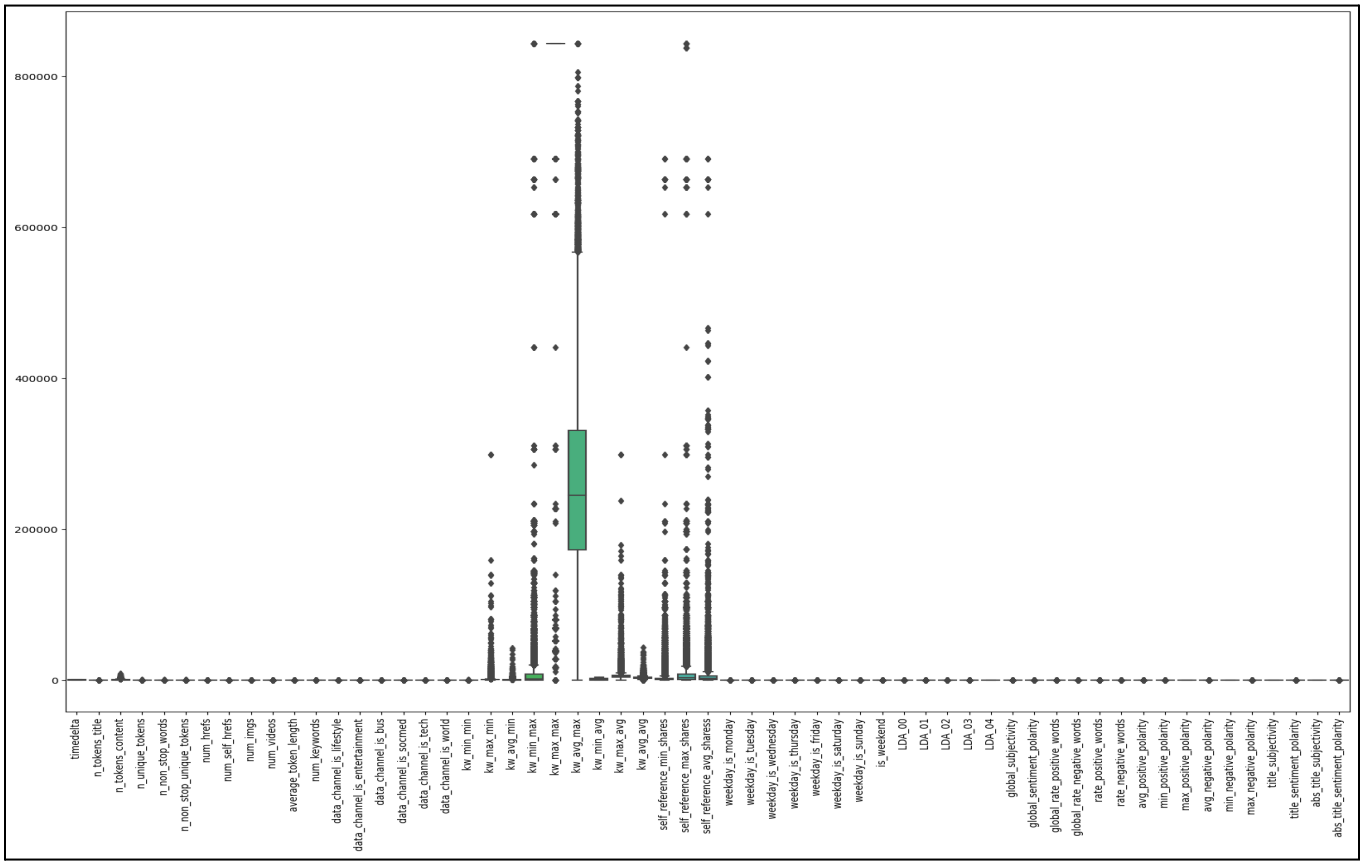


Saturday has maximum number of shares



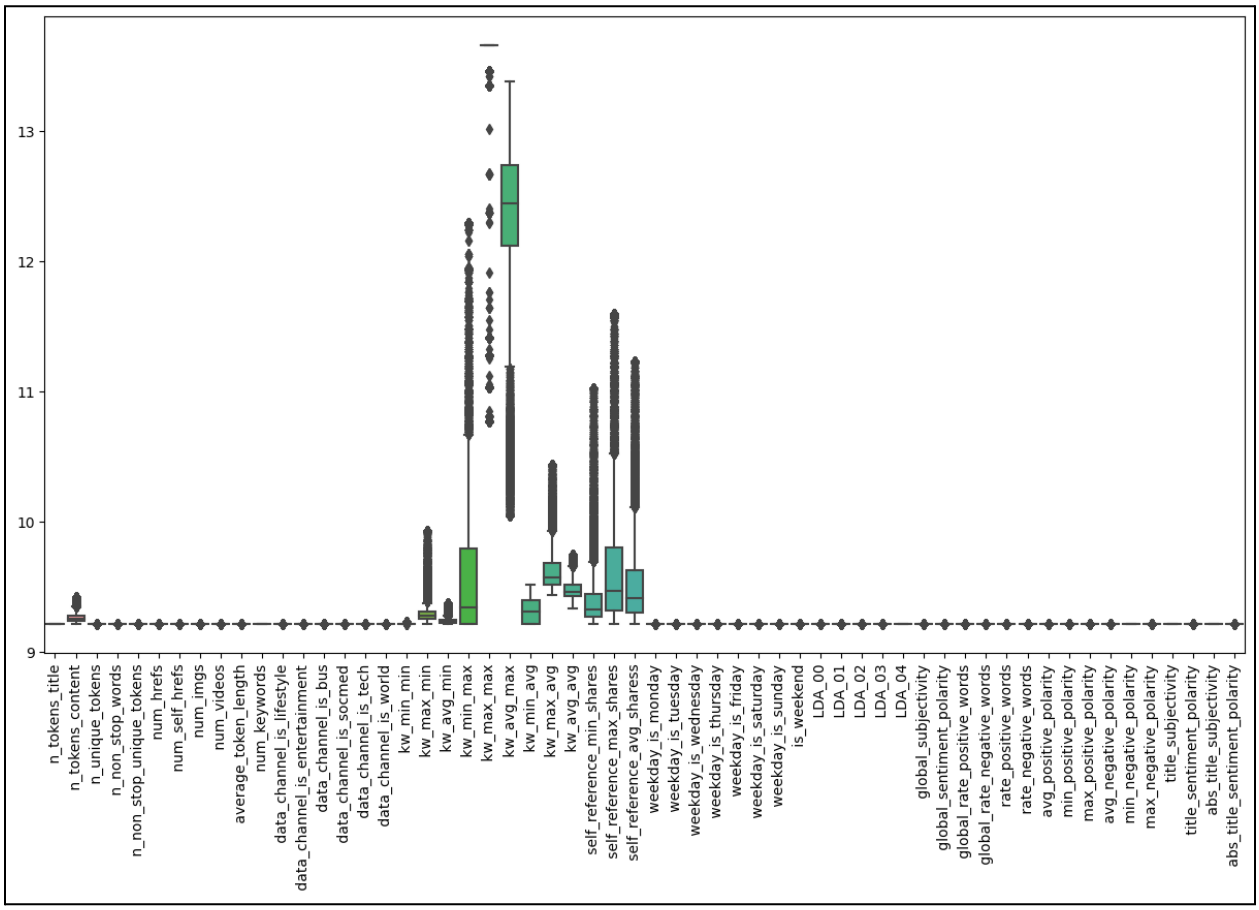
Max shares is when data channel is socmedia (Social Media)

Outlier Analysis



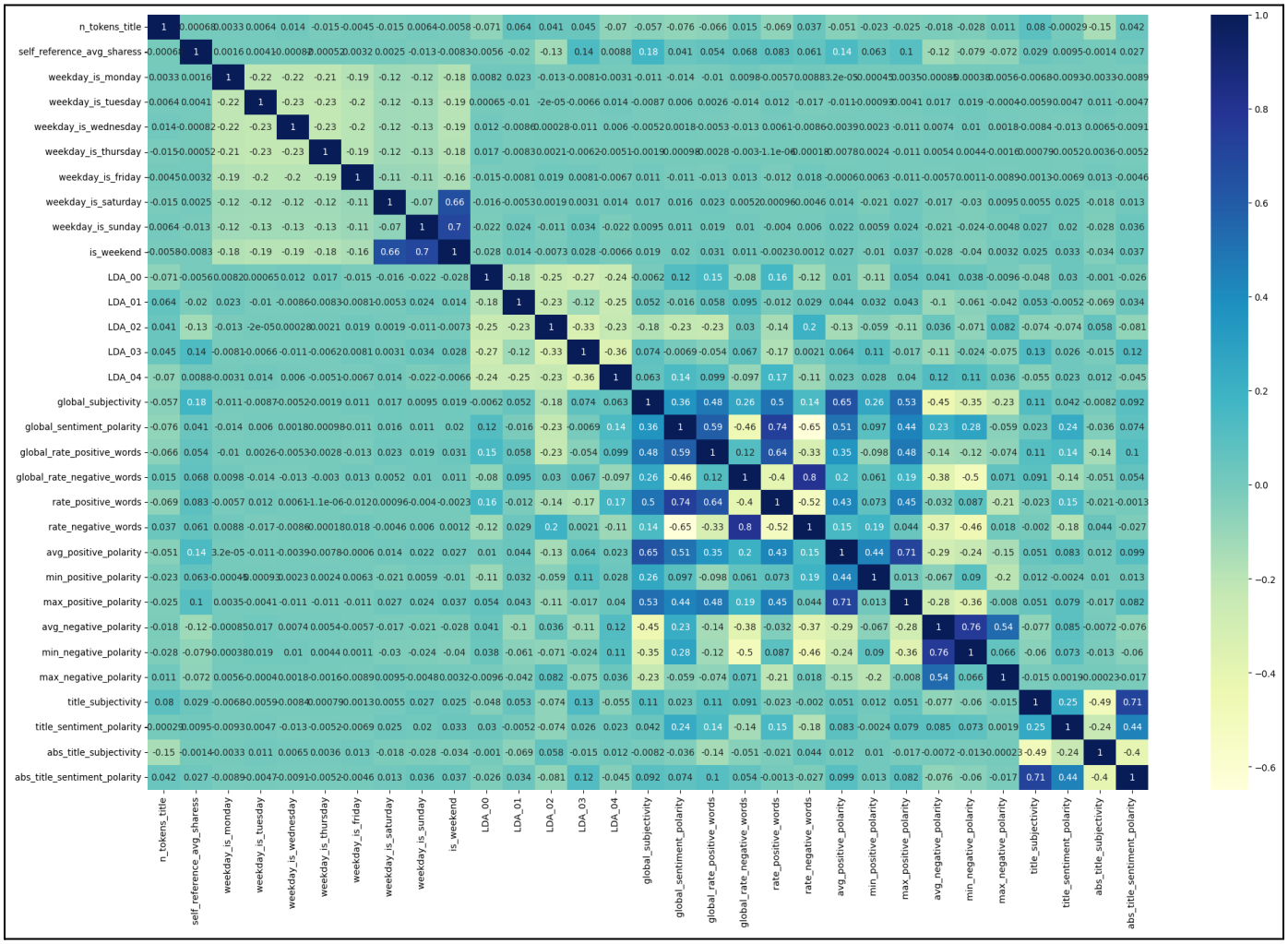
There are many outliers in the data, so we will remove them. We have converted negative values to positive values. Even after capping the values of columns to positive also few columns are not showing good results so dropping them.

```
# Treating Outlier
for col in X_train.columns:
    percentiles = X_train[col].quantile([0.01, 0.99]).values
    X_train[col][X_train[col] <= percentiles[0]] = percentiles[0]
    X_train[col][X_train[col] >= percentiles[1]] = percentiles[1]
```



Columns Dropped - ' kw_min_min', kw_max_min',kw_avg_min', 'kw_min_max', 'kw_max_max', kw_avg_max','kw_min_avg','kw_max_avg','kw_avg_avg','data_channel_is_world', self_reference_min_shares','self_reference_max_shares','data_channel_is_tech', data_channel_is_socmed','data_channel_is_lifestyle','data_channel_is_entertainment', data_channel_is_bus','data_channel_is_socmed','data_channel_is_tech', data_channel_is_world', num_keywords', average_token_length', num_videos','num_hrefs','num_self_hrefs','num_imgs', num_videos','n_non_stop_unique_tokens','n_non_stop_words','n_unique_tokens','n_tokens_content'

Looking at the correlation heat-map again -



Scaling - Serial vs Parallel Computation

After the earlier steps - in total we have 31 features now. We have used StandardScaler() to scale the data using both Scikit Learn and MLlib.
It standardizes features by removing the mean and scaling to unit variance. The standard score of a sample x is calculated as:

$$Z = \frac{x - \mu}{\sigma}$$

where μ is the mean of the training samples or zero if with_mean=False, and σ is the standard deviation of the training samples or one if with_std=False.

To prepare the data for MLlib functions and classes we have used Spark SQL DataFrame API - which has all the utilities and benefits of parallelization.

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)

train_x_pyspark = sqlContext.createDataFrame(X_train)

columns_names = list(X_train.columns)
from pyspark.ml.feature import StandardScaler as pyspark_scale
from pyspark.ml.feature import VectorAssembler
assembler = VectorAssembler().setInputCols(columns_names[:]).setOutputCol("features")
transformed_train_X = assembler.transform(train_x_pyspark)

%%time
scaler_pyspark = pyspark_scale(withMean=True, withStd=True,inputCol="features", outputCol="scaledFeatures")
scaler_pyspark_model = scaler_pyspark.fit(transformed_train_X.select("features"))
train_df_scaled_pyspark_X = scaler_pyspark_model.transform(transformed_train_X)
```

When standardizing the data, we also measure the time it takes to run, over ten trials. We note how the overhead to parallelize the data is greater than the computation itself. Hence, the z-score computation with scikit learn is faster than multiple partitions with pyspark(we have assumed 5 partitions).

Scaling Type	Time(in ms)
Serial	29.1
Parallel	73.4

Serial Computation - Models

Evaluation Metrics

These are the evaluation metrics we have used to evaluate our models on Test Data -

MAE (Mean Absolute Error) measures the average magnitude of errors between predicted and actual values.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

MSE (Mean Squared Error) measures the average of the squared differences between predicted and actual values.

$$\text{Mean Squared Error} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

RMSE (Root Mean Squared Error) is the square root of the average of the squared differences between predicted and actual values

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Baseline Model - Linear Regression with SGD (No Regularizer)

This is a basic regression model that assumes a linear relationship between the input features and the target variable. Compared to gradient descent, SGD uses an estimate of the gradient $g(x^k, w^k)$ to decide where the next step will go.

$x^{k+1} = x^k - \gamma^k\, g(x^k, w^k)$

Cost Function : $$J = \frac{1}{n}\sum_{i=1}^{n}(pred_i - y_i)^2$$

```
%%time
reg = SGDRegressor(max_iter=100, alpha=0.0, learning_rate='invscaling',
                   eta0=0.001, penalty = None, fit_intercept = True)
reg_model = reg.fit(X_train, y_train)
```

The Learning Rate is set to ***invscaling*** where the step size is given by $\eta = \eta_0/t^k$ (similar to what was discussed in HW-4) and $\eta_0 = 0.001$. There is no penalty term included in this run.

Average training time for 10 trials :

Trial	Training Time (in ms)
1	91.3
2	68.9
3	63.1
4	55.9
5	58.6
6	70.3
7	96.4
8	57.8

9	90.7
10	93.8
Average	74.68

Evaluation on Test Data

MAE	0.6687
MSE	0.7936
RMSE	0.8909

Linear Regression with SGD - L1 Regularization

In Lasso regression, a Norm - 1 penalty term is added for regularization. The cost function is given by :

L1 Regularization

$$\text{Cost} = \sum_{i=0}^{N}(y_i - \sum_{j=0}^{M} x_{ij}W_j)^2 + \lambda \sum_{j=0}^{M} |W_j|$$

```
%%time
reg = SGDRegressor(max_iter=100, alpha=0.0001, learning_rate='invscaling',
                   eta0=0.001, penalty = 'l1',fit_intercept = True)
reg_model = reg.fit(X_train, y_train)
```

The hyper-parameter is set to - 0.0001

The evaluation on Test Data is as follows

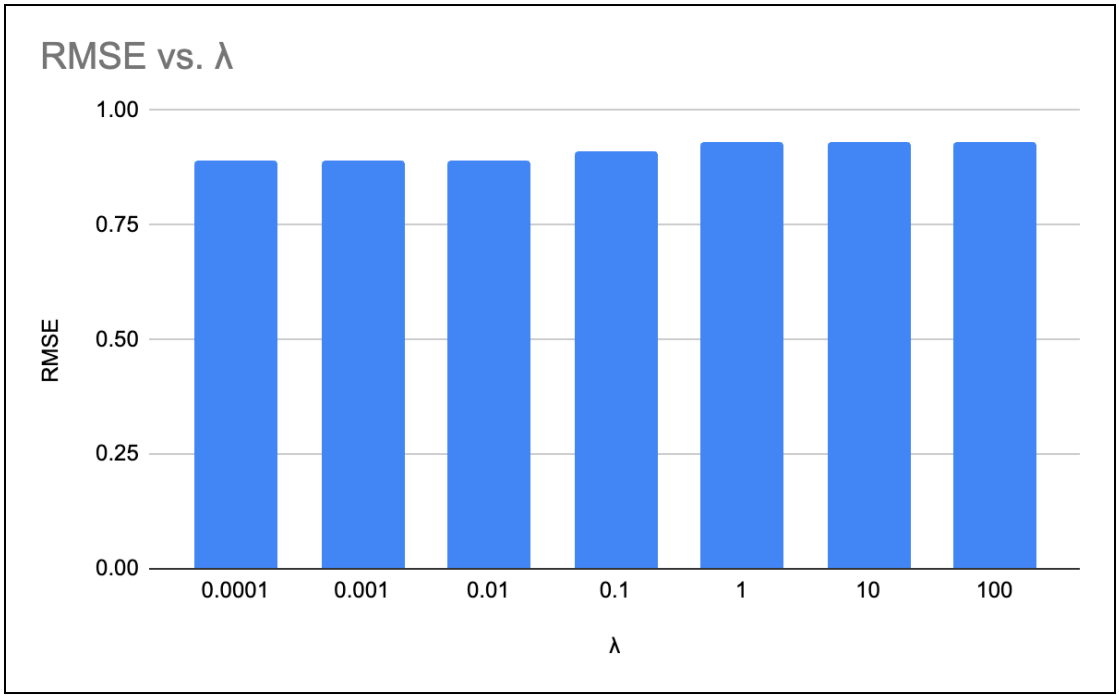
MAE	0.6682
MSE	0.7935
RMSE	0.8908

Average training time over 10 trails for λ = 0.0001 -

Trial	Training Time (in ms)
1	230
2	140

3	135
4	226
5	134
6	139
7	214
8	127
9	132
10	118
Average	159.5

Varying the λ value and plotting with the RMSE value -



Linear Regression with SGD - L2 Regularization

Similar to LASSO(L1 Regression), Ridge regression adds a penalty term to the loss function. However, this penalty term is based on the sum of the squared values of the model coefficients. Ridge regression is used to prevent overfitting and improve the generalization ability of the model.

L2 Regularization

$$\text{Cost} = \sum_{i=0}^{N} (y_i - \sum_{j=0}^{M} x_{ij} W_j)^2 + \lambda \sum_{j=0}^{M} W_j^2$$

Loss function

Regularization Term

```
%%time
reg = SGDRegressor(max_iter=100, alpha=0.0001, learning_rate='invscaling',
                   eta0=0.001,penalty = 'l2',fit_intercept = True)
reg_model = reg.fit(X_train, y_train)
```

The hyper-parameter is set to - 0.0001

The evaluation on Test Data is as follows

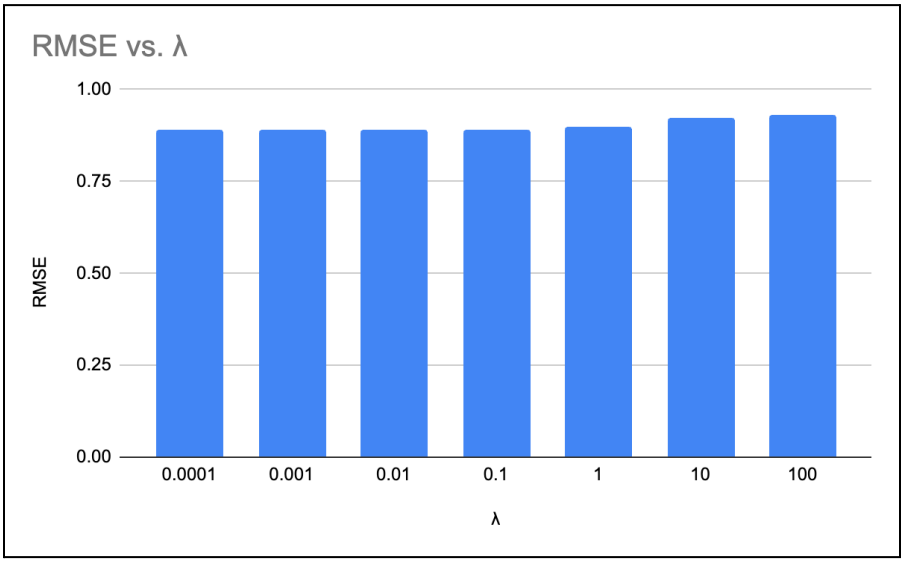
MAE	0.6693
MSE	0.7938
RMSE	0.891

Average execution time over 10 trails for = 0.0001 -

Trial	Training Time (in ms)
1	113
2	160
3	107
4	113
5	83.8
6	81
7	111
8	112
9	76.9
10	111

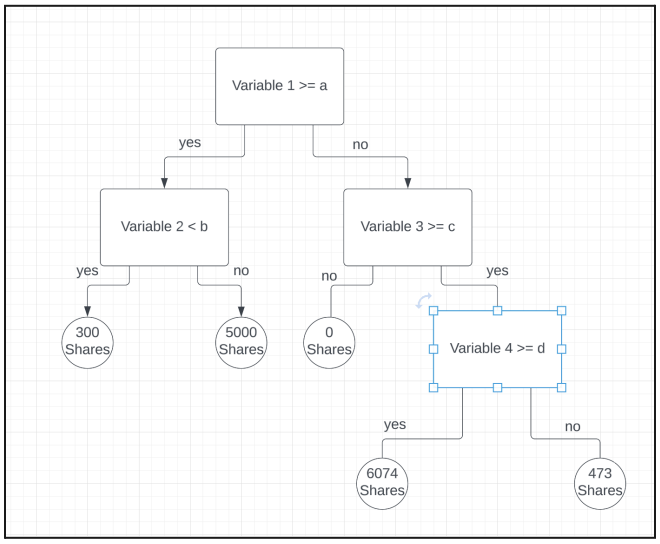
Average	106.87

Varying the λ value and plotting with the RMSE value -



Random Forest Regressor

Random Forest is a popular ensemble learning technique that combines multiple decision trees to make predictions. In our case we have used 25 trees, with depth until it prunes. Random Forest can handle nonlinear relationships between the input features and the target variable, and can be useful when there are many input features.



A representation of how the decision tree functions

```
%%time
from sklearn.ensemble import RandomForestRegressor
reg = RandomForestRegressor(n_estimators = 25,max_features = "auto", random_state=42,max_depth = 7)
reg.fit(X_train, y_train)
```

Training Time from 10 Trials -

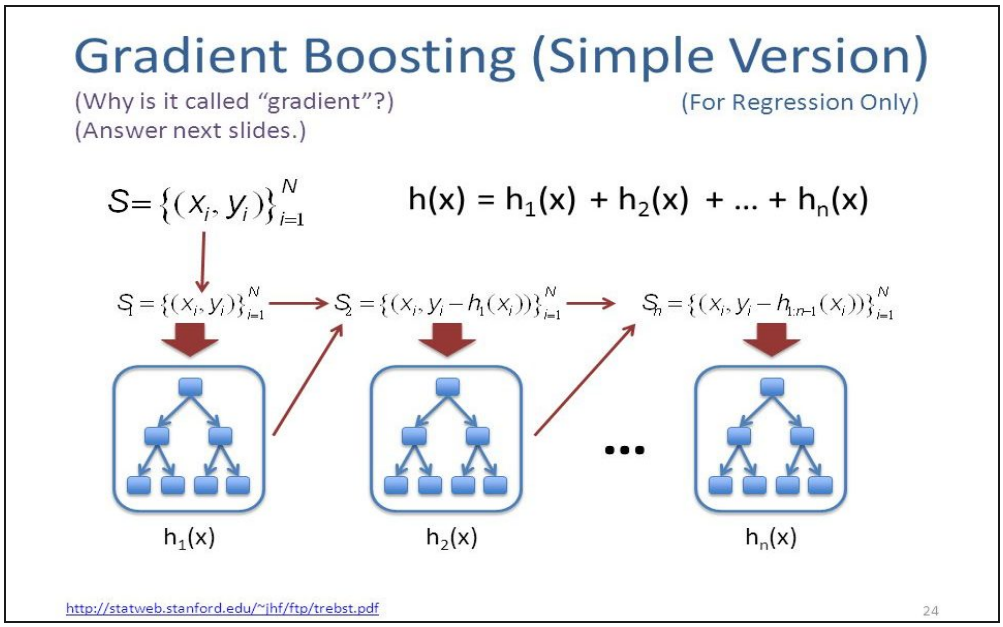
Trial	Training Time (in s)
1	7.91
2	7.77
3	7.61
4	7.65
5	7.91
6	7.11
7	7.04
8	7.13
9	7.6
10	7.48
Average	7.521

Evaluation of Test Data -

MAE	0.6659
MSE	0.7822
RMSE	0.8844

Gradient Boosting Regressor

Gradient Boosting is another ensemble learning technique that combines multiple weak learners to make predictions. The main difference between random forests and gradient boosting lies in how the decision trees are created and aggregated. Unlike random forests, the decision trees in gradient boosting are built additively; in other words, each decision tree is built one after another.



Training Time - Over 10 trials :

Trial	Training Time (in s)
1	22
2	20.3
3	20.4
4	21
5	21.3
6	22.5
7	20.7
8	20.9
9	22.3
10	20.1
Average	21.15

Evaluation of Test Data -

MAE	0.6592
MSE	0.7736
RMSE	0.8796

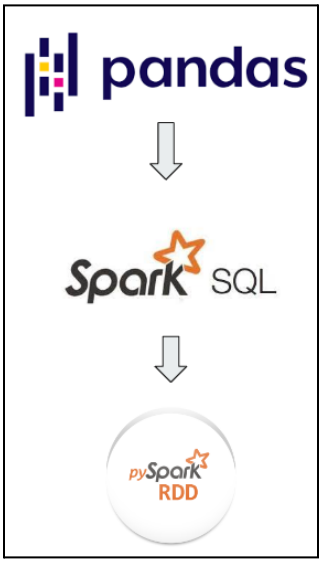
Model Evaluation and Time Statistics

Algorithms	MAE	MSE	RMSE
Linear Regression with SGD (No Penalty)	0.6687	0.7936	0.8909
L1 Regularization	0.6682	0.7935	0.8908
L2 Regularization	0.6693	0.7938	0.891
Random Forest Regressor	0.6659	0.7822	0.8844
Gradient Boosting Regressor	0.6592	0.7736	0.8796

Algorithms	Time
Linear Regression with SGD (No Penalty)	74.68 ms
L1 Regularization	159.5 ms
L2 Regularization	106.87 ms
Random Forest Regressor	7.521 s
Gradient Boosting Regressor	21.15 s

Parallel Implementation

Data Preparation



For parallel implementation we need to prepare the data in RDD format since we will be using MLLib RDD based library functions to implement the models in parallel.

All the data till this point is in pandas dataframe. We initialized a spark SQL context to convert the pandas dataframe to Spark SQL Dataframe.

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
```

```
train_df_x = sqlContext.createDataFrame(X_train)
train_df_y = sqlContext.createDataFrame(pd.DataFrame(y_train))
test_df_x = sqlContext.createDataFrame(X_test)
test_df_y = sqlContext.createDataFrame(pd.DataFrame(y_test))
```

After Spark SQL Dataframe we converted it to rdd.

```
from pyspark.ml.feature import VectorAssembler
columns_names = train_df_x.columns
assembler = VectorAssembler().setInputCols(columns_names).setOutputCol("features")
transformed_train_df = assembler.transform(train_df_x)
transformed_test_df = assembler.transform(test_df_x)
transformed_train_df.count()
```

```
33697
```

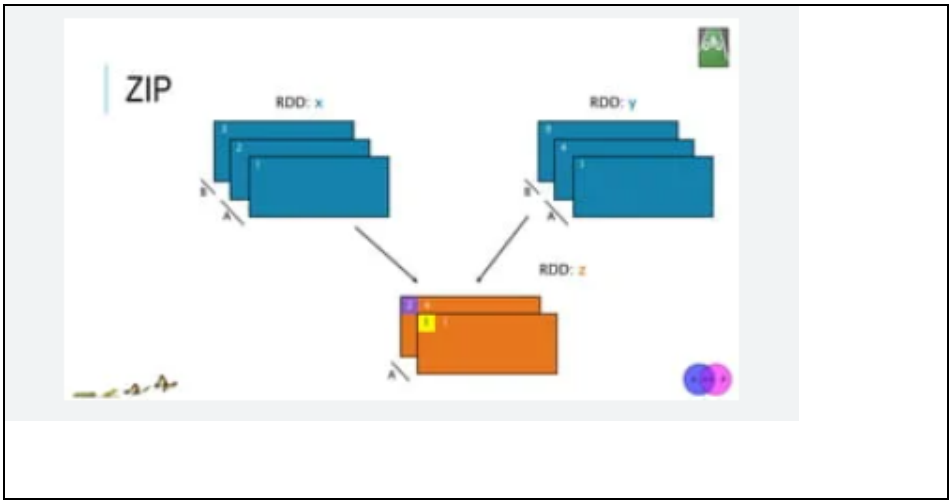
```
train_rdd_x = transformed_train_df.select(transformed_train_df["features"]).rdd
train_rdd_x = train_rdd_x.map(lambda x: list(x[0])).repartition(10)
```

```
test_rdd_x = transformed_test_df.select(transformed_test_df["features"]).rdd
test_rdd_x = test_rdd_x.map(lambda x: list(x[0])).repartition(10)
```

```
train_rdd_y = train_df_y.select(train_df_y[" shares"]).rdd
train_rdd_y = train_rdd_y.map(lambda x: x[0]).repartition(10)
```

```
test_rdd_y = test_df_y.select(test_df_y[" shares"]).rdd
test_rdd_y = test_rdd_y.map(lambda x: x[0]).repartition(10)
```

The data in the rdd is converted to LabelPoint class - to prepare it for input into the MLlib models -



↓

```
LabeledPoint(0.0, [0.0, 1.0]),
LabeledPoint(1.0, [1.0, 0.0]),
```

```
train_rdd = train_rdd_y.zip(train_rdd_x)
test_rdd = test_rdd_y.zip(test_rdd_x)
train_rdd.count()
```

```
33697
```

```
train_rdd_lp = train_rdd.map(lambda x : LabeledPoint(x[0], x[1]))
```

```
train_rdd_lp = train_rdd_lp.repartition(5)
```

```
test_rdd = test_rdd.repartition(5)
```

All further computations are done on 5 partitions

Linear Regression with SGD (No Regularizer)

```
%%time
lrm = LinearRegressionWithSGD.train(train_rdd_lp, iterations=100, step=1, intercept=True)
```

```
CPU times: user 103 ms, sys: 6.42 ms, total: 109 ms
Wall time: 11.7 s
```

```
%%time
test_preds = lrm.predict(test_rdd.map(lambda x: x[1]))
```

```
CPU times: user 898 µs, sys: 891 µs, total: 1.79 ms
Wall time: 6.02 ms
```

```
rdd_eval(test_rdd,lrm)
```

```
MAE:  0.6692725420273354
MSE:  0.7941914432415349
RMSE:  0.8911741935455352
PythonRDD[148] at RDD at PythonRDD.scala:53
```

Training Time Over 10 Trials -

Trial	Training Time (in ms)
1	109
2	59
3	42.2
4	44
5	47.8
6	54.5
7	54.2
8	38.6
9	56.1
10	49.7
Average	55.51

Linear Regression with SGD - L1 Regularization

```
%%time
lrm = LassoWithSGD.train(train_rdd_lp,initialWeights=np.random.rand(31), iterations=100,step=1, intercept=True,regParam = 0.0001)
```

```
CPU times: user 31.1 ms, sys: 5.36 ms, total: 36.4 ms
Wall time: 2.3 s
```

```
%%time
test_preds = lrm.predict(test_rdd.map(lambda x: x[1]))
```

```
CPU times: user 340 µs, sys: 1e+03 µs, total: 1.34 ms
Wall time: 1.38 ms
```

```
rdd_eval(test_rdd,lrm)
```

```
MAE:  0.6765556117625854
MSE:  0.808130262753798
RMSE:  0.8989606569554632
PythonRDD[2358] at RDD at PythonRDD.scala:53
```

λ = 0.0001

Training Time Over 10 Trials -

Trial	Training Time (in ms)
1	77.7
2	41.9
3	54.1
4	50.7
5	32.3
6	36.4
7	44.8
8	49.1
9	40.9
10	36.4
Average	46.43

Linear Regression with SGD - L2 Regularization

```
%%time
lrm = RidgeRegressionWithSGD.train(train_rdd_lp,initialWeights=np.random.rand(31),iterations=100,step=1, intercept=True,regParam = 0.0001)
```

```
CPU times: user 33.5 ms, sys: 8.59 ms, total: 42.1 ms
Wall time: 2.42 s
```

```
%%time
test_preds = lrm.predict(test_rdd.map(lambda x: x[1]))
```

```
CPU times: user 1.56 ms, sys: 0 ns, total: 1.56 ms
Wall time: 8.6 ms
```

```
rdd_eval(test_rdd,lrm)
```

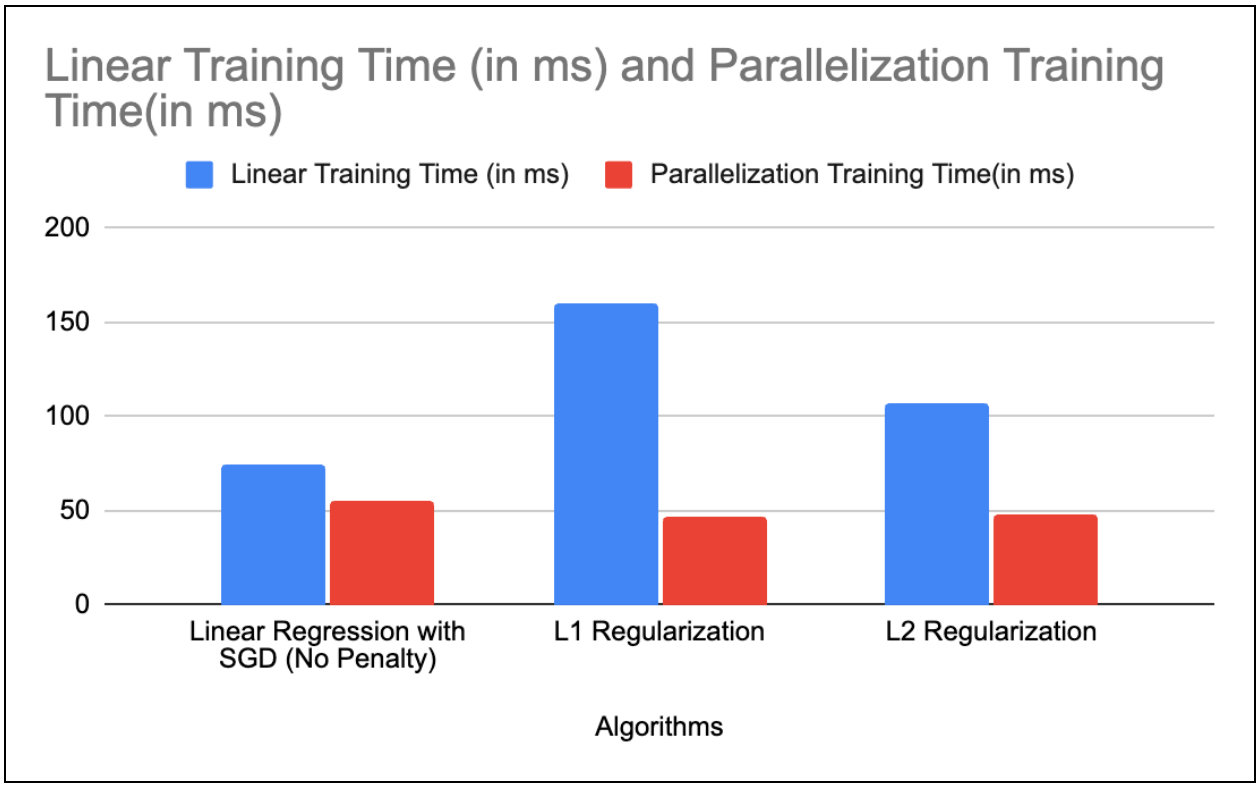
```
MAE:  0.6625171107024951
MSE:  0.8079316435282403
RMSE:  0.8988501785771866
PythonRDD[324] at RDD at PythonRDD.scala:53
```

λ = 0.0001

Training Time Over 10 Trials -

Trial	Training Time (in ms)
1	61.1
2	42.7
3	51.1
4	50.7
5	54.6
6	47.3
7	44.8
8	49.1
9	40.9
10	36.4
Average	47.87

Comparison with Serial Computation



Random Forest Regressor

Training Time - Over 10 Trials

Trial	Training Time (in ms)
1	111
2	98
3	98.1
4	84.2
5	90.8
6	96.4
7	98.7
8	106
9	135
10	150
Average	106.82

```
%%time
rfm = RandomForest.trainRegressor(train_rdd_lp,{}, numTrees = 25, seed=42,maxDepth = 7,featureSubsetStrategy = "all",maxBins = 31)

CPU times: user 130 ms, sys: 19.1 ms, total: 150 ms
Wall time: 21.9 s

%%time
test_preds = rfm.predict(test_rdd.map(lambda x: x[1]))

CPU times: user 5.51 ms, sys: 984 µs, total: 6.5 ms
Wall time: 45.7 ms

rdd_eval(test_rdd,rfm)

MAE:  0.6640672080264939
MSE:  0.7814910051554627
RMSE:  0.8840197990743548
PythonRDD[401] at RDD at PythonRDD.scala:53
```

Gradient Boosting Regressor

Trial	Training Time (in ms)
1	761
2	573
3	552
4	629
5	609
6	558
7	626
8	614
9	655
10	834
Average	641.1

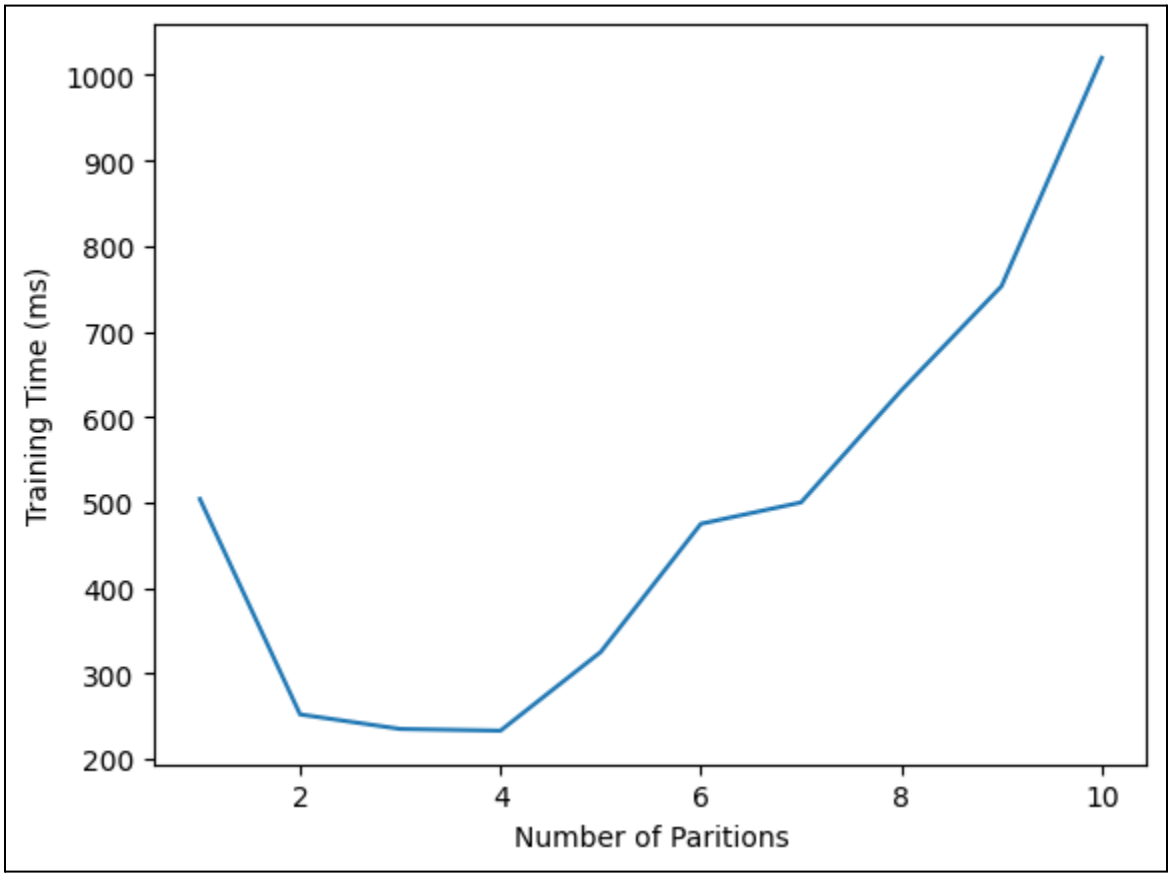
```
%%time
gbt = GradientBoostedTrees.trainRegressor(train_rdd_lp, {}, numIterations=100)

CPU times: user 700 ms, sys: 134 ms, total: 834 ms
Wall time: 1min 25s

%%time
test_preds = gbt.predict(test_rdd.map(lambda x: x[1]))

CPU times: user 5.58 ms, sys: 1 ms, total: 6.58 ms
Wall time: 28.6 ms

rdd_eval(test_rdd,gbt)

MAE:  0.657102309830288
MSE:  0.7722342162387416
RMSE:  0.8787685794557868
PythonRDD[8537] at RDD at PythonRDD.scala:53
```

Comparison with Serial Computations -

Algorithms	Linear Training Time **(in sec)**	Parallelization Training Time**(in ms)**
Random Forest Regressor	7.521	106.82
Gradient Boosting Regressor	21.15	641.1

There is a difference in the scale of training time. So, we may not be comparing apples to apples and even after further investigation - we were not able to completely attune the two models we have used from sklearn and MLLib. However, there is considerable improvement in model training time - regardless.

Analysis with different Partition Numbers for best model - Gradient Boosted Trees Regressor



Minimum Training Time is for 4 partitions

K-Fold Cross Validation

We performed K-fold Cross Validation on the data with 4 folds using Gradient Boosting Regression and noted the RMSE for each fold -

Fold - 1	0.8803429081
Fold - 2	0.8848325477
Fold - 3	0.9059120016
Fold - 4	0.9137372907

Conclusion

In this project - we have considered a classic regression problem to predict the number of shares of New Articles based on input features. Karan Desai was responsible for Scaling, and Linear Regression without Regularization and Linear Regression with L1 Regularization. Ishan Palit worked on Data Pre-Processing , Linear Regression with L2 Regularization and K-fold Cross Validation. Dhruvi Gajjar was responsible for Random Forest Regressor and Gradient Boosting Regressor. The comparisons and contrasts were worked on as a team.

From all the models we evaluated on the test data, we considered RMSE as the main metric for comparison.

Algorithms	**MAE**	**MSE**	**RMSE**
Linear Regression with SGD (No Penalty)	0.6687	0.7936	0.8909
L1 Regularization	0.6682	0.7935	0.8908
L2 Regularization	0.6693	0.7938	0.891
Random Forest Regressor	0.6659	0.7822	0.8844
Gradient Boosting Regressor	0.6592	0.7736	0.8796

Gradient Boosting Regressor performed the best followed by Random Forest Regressor.
In the future scope, we can perform more hyperparameter tuning in both the algorithms mentioned and also work on implementing cache() and unpersist() in the pyspark code to improve training times.

References

https://archive.ics.uci.edu/ml/datasets/online+news+popularity

chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://core.ac.uk/download/pdf/55638607.pdf

https://spark.apache.org/docs/2.2.0/mllib-guide.html

https://scikit-learn.org/stable/supervised_learning.html#supervised-learning

https://pandas.pydata.org/docs/