Using Twitter Data to Predict Bitcoin’s Price

In recent years, cryptocurrencies have seen a massive increase in value. This is partially due to many advocates touting them as replacements to current currencies. Others see crypto as a diversification of investment portfolios or a way to generate a quick profit. The increase in the prices of cryptocurrencies like Bitcoin, though, masks how volatile the currency’s price is, which makes it a precarious entity to invest in.

Regardless, there has been a lot of excitement about Bitcoin, particularly on social media platforms like Twitter. On Twitter, the idea of the wisdom of the crowd is evident, as large groups of people can engage with each other on a current event and reach a broad consensus. This raises an interesting question: can we use Twitter to make better sense of the volatility of Bitcoin’s price?

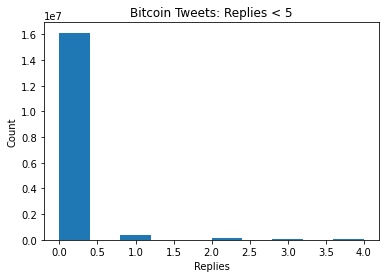
To investigate this, we chose to analyze 2 datasets from Kaggle: one containing text from 40,000,000 [Bitcoin tweets](https://www.kaggle.com/alaix14/bitcoin-tweets-20160101-to-20190329) from January 2016 to March 2019, and the other containing historical [Bitcoin prices](https://www.kaggle.com/mczielinski/bitcoin-historical-data) from January 2012 to March 2021. In our project, we analyzed and visualized certain characteristics of our data, then used Apache Spark and MLLib to wrangle and model our data to attempt to predict changes in Bitcoin prices.

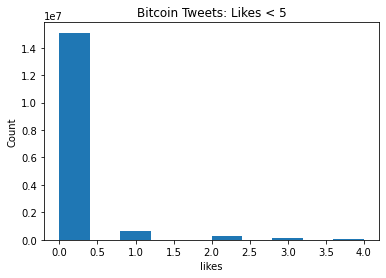
## Exploratory Data Analysis

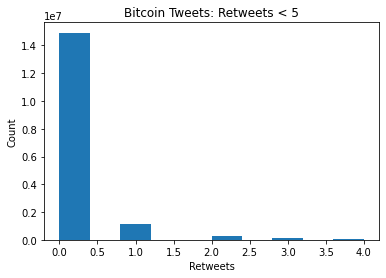
Starting with the tweets, we took a look at the length of the tweets included in the dataset:

Since the maximum length of tweets used to be 140 characters before Twitter raised the character limit to [280 characters](https://techcrunch.com/2018/10/30/twitters-doubling-of-character-count-from-140-to-280-had-little-impact-on-length-of-tweets/) in 2017-2018, it is unsurprising that 140 characters appears to be the most common tweet length, with tweets above 140 characters becoming more rare (i.e., longer tweets were allowed mid-way through the data collection period).

In terms of standard metrics of tweets, the 3 most commonly cited are replies, likes, and retweets. Visualizing these 3 metrics, though, we see a similar trend:

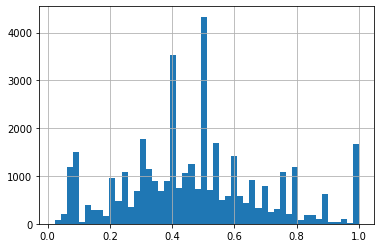
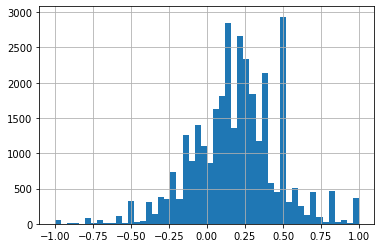




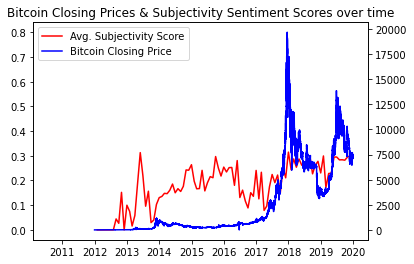
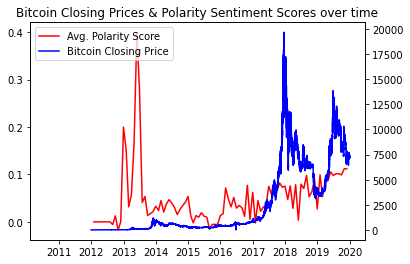


We note that all 3 metrics have nearly identical distributions; namely, almost all tweets have 0 replies, likes, and retweets, which makes sense given that the majority of Twitter users’ tweets have little-to-no engagement whereas a small proportion of influential Twitter accounts (such as celebrities, CEOs, etc.) are highly-interacted with. In fact, looking at the distribution of the metrics < 5, we find that the drop off between 0 and 1 interaction is huge. This seems to suggest that the average number of replies/likes/retweets on a given day is not particularly useful, as it will likely be very close to 0. Instead, the maximum may be more useful, as it indicates the most influential voice tweeting about Bitcoin on a given day.

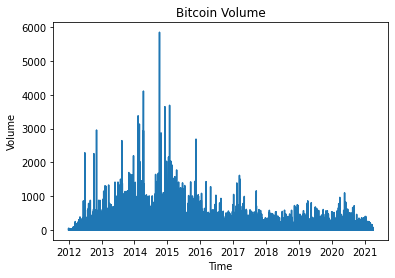
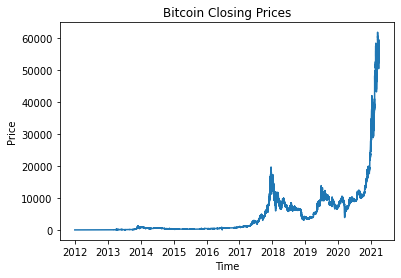
Moving onto the sentiment analysis of the tweets (see next section to see how this was done), it appears as though the polarity, or the positive/negative sentiment, of the Bitcoin tweets leans more towards the positive end, with a median of around 0.25. This is consistent with Bitcoin’s general increase of value over time. The subjectivity of each tweet seems to be somewhat evenly distributed.



There also seems to be some correlation between polarity and Bitcoin price, as well as subjectivity and Bitcoin price, across time, as seen in the following visualizations:



With regard to historical Bitcoin prices, it is not surprising to find that Bitcoin prices, which were once very low in the past, have become incredibly high, especially in the past few years. The number of Bitcoins that are being transacted appear to be relatively constant/slightly decreasing.



## Data Wrangling

One of the major difficulties that we had in this project was working with the large datasets. It can be rather difficult to handle tens of millions of rows of data, so we spent time considering the best way to process the data for use in our models. Because of the large size of the datasets, we chose to upload the files to an Amazon S3 bucket, so that local storage/RAM would not be used to download the data. We also opted to use Apache Spark through Amazon EMR clusters, which allowed us to use distributed computation to more quickly process the data.

We first set up the schema for our csv files and imported them as Spark dataframes:

| from pyspark.sql.types import \*  # Schema for bitcoin-tweets.csv tweets\_schema = StructType(  [StructField('id', FloatType(), True),  ...  StructField('text', StringType(), True),  ] )  # Schema for bitcoin-prices.csv prices\_schema = StructType(  [StructField('Timestamp', IntegerType(), True),  ...  StructField('Weighted\_Price', FloatType(), True),  ] )  # Import csv files from S3 bucket as Spark Dataframes tweets\_sdf = spark.read.option('delimiter', ';') \  .csv("s3://cis-545-project-bitcoin-files/bitcoin-tweets.csv",   schema = tweets\_schema, header = True) prices\_sdf = spark.read.csv(  "s3://cis-545-project-bitcoin-files/bitcoin-prices.csv",   schema = prices\_schema, header = True) |
| --- |

Aside from standard wrangling steps (e.g., dropping null rows), one of the more challenging things that we had to use was applying Python functions to our Spark dataframes using udfs. Udfs proved to be incredibly useful for us in our project, as many tasks involving the tweet content (cleaning the tweets, sentiment analysis, etc.) used udfs. The following excerpt describes the udfs used to calculate polarity and subjectivity of the cleaned tweets:

| # Determine polarity, subjectivity of the tweets def polarity\_score(tweet):  text = textblob.TextBlob(tweet)  return text.sentiment.polarity  def subjectivity\_score(tweet):  text = textblob.TextBlob(tweet)  return text.sentiment.subjectivity  polarity\_udf = udf(lambda x: polarity\_score(x), FloatType()) subjectivity\_udf = udf(lambda x: subjectivity\_score(x), FloatType())  tweets\_sentiment\_sdf = tweets\_lang\_sdf.select('Timestamp', 'Replies',  'Likes', 'Retweets', 'Text',  polarity\_udf('Text').alias('Polarity'),  subjectivity\_udf('Text').alias('Subjectivity')) |
| --- |
| tweets\_sentiment\_sdf.show(5) |

After extraneous columns (e.g., url, fullname) were removed and desired columns (e.g., polarity) were added to the dataframes for Bitcoin tweets and prices, due to the size of the Bitcoin tweet dataset, we elected to randomly sample rows from the dataset. We then merged the tweet and price dataframes together on the Date column to form a final dataframe. This dataframe includes the following columns:

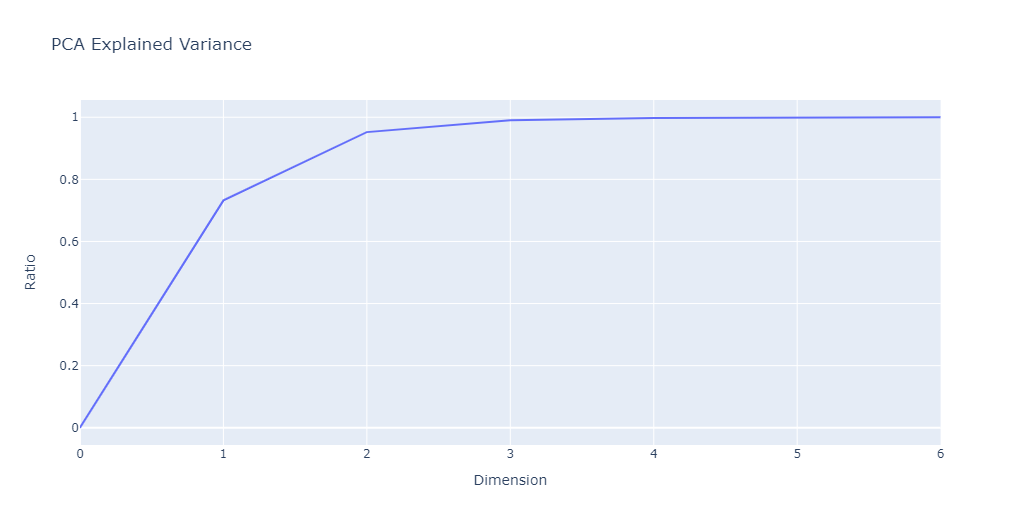
* Date
* Volume (of Bitcoin)
* Replies
* Likes
* Retweets
* Polarity
* Subjectivity
* Price - Value to be predicted

## Modeling

Continuing in the same vein as the data wrangling, as we are working with large datasets, we chose to use MLLib to produce our models as opposed to Scikit-learn or PyTorch. MLLib is Apache Spark’s machine learning library, and can more easily be scaled to big data.

Before this analysis, though, the dataframe that we have must be transformed accordingly and split into training and test sets, as seen below:

| ...  from pyspark.ml.feature import VectorAssembler from pyspark.ml import Pipeline  # Create a feature vector, transform the dataframe assembler = VectorAssembler(inputCols = feature\_columns, outputCol = "features") pipeline = Pipeline(stages = [assembler]) pipeline\_model = pipeline.fit(final\_sdf) transformed\_sdf = pipeline\_model.transform(final\_sdf) transformed\_sdf.show(5)  # Split dataset to train, test train\_sdf, test\_sdf = transformed\_sdf.randomSplit([0.8, 0.2], seed = 1) |
| --- |

After selecting the feature columns, of which we have 6 (excluding Price, which is the label, and Date), we chose to perform dimensionality reduction using PCA, as we hypothesized that certain columns (e.g., Replies) would have minimal explanatory power or have overlap with other features. By performing PCA, we found that 73.23% of the variance was explained by the 1st component, and 99.73% by the 4th one. As such, we let 4 be the number of dimensions for the PCA features.

In terms of models, we tested 4 major ones, after having used PCA to transform our train and test data:

1. Linear Regression
2. Linear Regression with Regularization
3. Random Forest Regression
4. Random Forest Regression with Hyperparameter Tuning

Our first model was a simple Linear Regression, which was run without any regularization or tuning. This model performed decently well, yielding an accuracy of 0.3878 and an RMSE value of 4.1120 on the testing data.

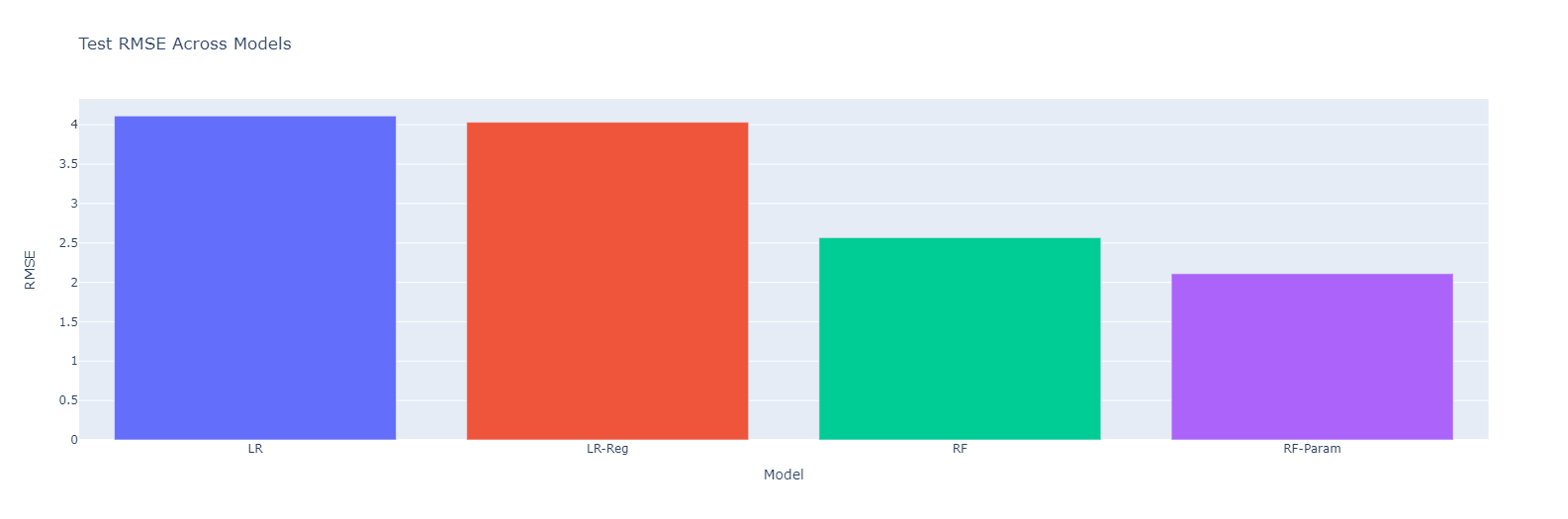
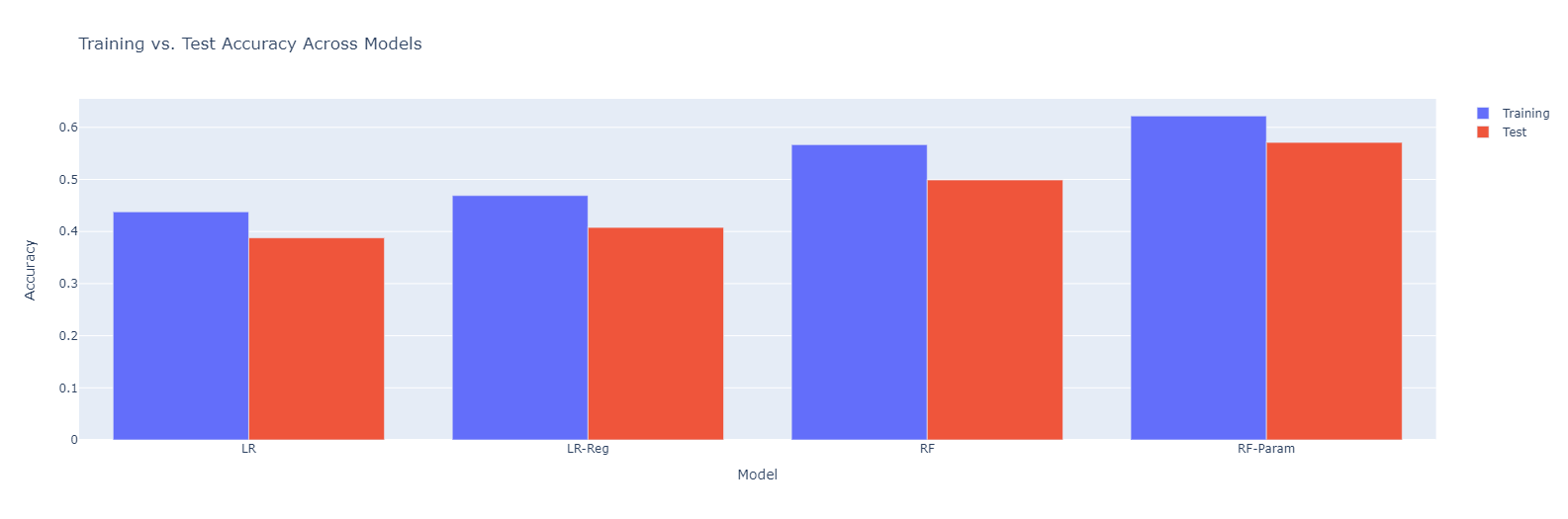
Our second model was Linear Regression with Regularization, during which we tested different regularization techniques including LASSO (L1), Ridge (L2), and Elastic Net (L1+L2). This was done by using a Parameter Grid and Spark’s CrossValidator() to tune the model’s hyperparameters, and from this process we found that the best regParam = 0.1 and the best elasticNetParam = 0.5, indicating that Elastic Net (L1+L2) had the best performance (LASSO has elasticNetParam = 1 and Ridge has elasticNetParam = 0). Our selected Linear Regression with Regularization model yielded an accuracy of 0.4078 and an RMSE value of 4.0319 on the testing data, which was only marginally better than our first model.

Our third model was a Random Forest Regression, which is an ensemble learning method that uses bagging to aggregate multiple decision trees. This model yielded a training accuracy of 0.5668 and a testing accuracy of 0.4990, with this larger discrepancy suggesting that it was slightly overfit on the training data.

To address this observation, our final model was a Random Forest Regression with Hyperparameter Tuning. We utilized a Parameter Grid and Spark’s CrossValidator() to test a range of values for each relevant hyperparameter (numTrees & maxDepth) of the model. This model yielded a training accuracy of 0.6219 and a testing accuracy of 0.5710, meaning that it performed the best out of all our models and was able to reduce the overfitting observed in our initial Random Forest model. The code for the hyperparameter tuning is as follows:

| ...  paramGrid = ParamGridBuilder() \  .addGrid(numTrees, [5, 10, 20, 30, 40, 50]) \  .addGrid(maxDepth, [5, 10, 15, 20]) \  .build()  crossVal = CrossValidator() \  .setEstimator(rf\_model) \  .setEvaluator(rf\_evaluator) \  .setEstimatorParamMaps(paramGrid) \  .setNumFolds(4) ) ... |
| --- |

The two graphs below illustrate the training/test accuracies and test RMSE values for each of our models.



## Challenges Faced & Future Steps

While our models performed relatively well given the complexity of the task, there are still various improvements and future avenues of investigation to look into.

For one, we were limited by the timeframe in which the Twitter dataset was collected, as the Bitcoin tweets were collected between January 2016 and March 2019. This misses out on Bitcoin’s recent surge in price (as can be seen in the previous plot of Bitcoin price over time). This information would’ve been valuable to account for in our models and would’ve made the models more robust. As a result, collecting more tweets after March 2019 would be a great way to further validate our results and improve our models in the future.

Additionally, there are other features that could’ve been considered in our analysis. For example, in our current models, we performed sentiment analysis on the tweets and used those numerical scores as inputs into our model. Another approach that we could’ve taken is to look at which words are used most frequently in tweets. This is because certain words which appear more often in tweets could indicate a rise in Bitcoin popularity, or vice-versa, which may have been useful in predictions.

For our modelling, we opted to use MLLib with Apache Spark for machine learning to deal with the large datasets. In the future, other available MLLib regression models (e.g. logistic regression) could be experimented with to see if they are more effective than the current tested models. We could also work with deep learning and neural networks, either by using PyTorch or Apache MXNet, which places an emphasis on distributed computing, in an attempt to create more predictive models.