Regime Trading Using the Wasserstein Metric

Ru Han Wang, Justin Cheung, Hari Suresh, and Cody Briggs
UC Davis, Davis, CA, USA

Abstract

The problem that we seek to solve as a result of this study relates to the timely detection of distinct market regimes for large cap U.S. equity assets (specifically those that are part of the SP 500 index), and whether or not these regimes can be used to design a profitable automated trading strategy. To do this, we seek to use a signal known as the "Wasserstein Metric" along with a modified version of the k-means clustering algorithm to identify "bullish" or "bearish" market regimes, and then subsequently place trades according to these regimes (going long equities during "bullish" regimes and going short equities during "bearish" regimes). In the process of identifying market regimes, we also determine if we can increase the computational speed of the k-Means clustering algorithm without altering its core functionality, as this is crucial to being able to identify market regimes in a timely manner to place automated trades before the regime shifts. A common problem with existing methods of identifying market regimes is that they take a significant amount of time to make a classification, by which time the regime in question has passed (and the trading opportunity no longer exists). Our study shows that the usage of the Wasserstein metric along with a modified k-means algorithm is timely and effective in identifying "bullish" and "bearish" market regimes a significant portion of the time, and that our automated trading strategy based off of these identified regimes is profitable relative to investing in the broader market (proxied by the returns of the SP 500).

1 Introduction

1.1 The concept of market regimes in the U.S. equity market An equity market regime can be thought of as a cluster of "persistent" market conditions that broadly impacts the prices of equity assets. In the case of our study, regimes in the U.S. equity market are classified based on the volatility (magnitude of price movements) and momentum (broader directional trend in price movements) of the constituents of the SP 500. In particular, our study seeks to identify whether the market is in one of four regimes: high volatility negative momentum, high volatility positive momentum, low volatility positive momentum, and low volatility negative momentum. Regimes with positive momentum

would be classified as being "bullish" for U.S. large cap equity returns, while regimes with negative momentum would be classified as being "bearish" for U.S. large cap equity returns.

1.2 Wasserstein's Metric Introduced by Russian mathematician Leonid Vaserstein in 1969, Wasserstein's metric, which is commonly used in optimal transport problems, is thought to represent the distance between two probability distributions (it is also used to identify the minimum energy required to move from one probability distribution to another). For vectors, Wasserstein distance of order p can be described as the mean absolute power difference between two vectors, sorted by increasing values.

$$W_p(u,v)^p = \sum_{i=1}^{N} \int_{(i-1)/N}^{i/N} |F_u^{-1}(z) - F_v^{-1}(z)|^p dz = \frac{1}{n} \sum_{i=1}^{N} |a_i - b_i|^p$$

For instance, consider the following example:
Given two vectors u and v:
$\overrightarrow{u} = [-0.3, 0.5, 0.9, -0.2]$
$\overrightarrow{v} = [0.4, -0.1, -0.5, 0.7]$

The p-Wasserstein difference between the vectors u and v can be determined by first sorting the two vectors into a and b, and then taking the average of the absolute differences between corresponding elements of either vector.

In the above case, the 1-Wasserstein distance would be equal to the following:
$\overrightarrow{a} = [-0.3, -0.2, 0.5, 0.9]$
$\overrightarrow{b} = [-0.5, -0.1, 0.4, 0.7]$

$1 - \text{Wasserstein distance} = \frac{|-0.3-(-0.5)|+|-0.2-(-0.1)|+|0.5-0.4|+|0.9-0.7|}{4}$
$1 - \text{Wasserstein distance} = \frac{0.2+0.1+0.1+0.2}{4}$
$1 - \text{Wasserstein distance} = 0.15$

Wasserstein's metric is useful for determining market regimes because it eliminates the effect of sequential orders (which can make identification of clusters of persistent market

conditions easier).

1.3 K-means clustering algorithm K-means clustering is an algorithm where the input dataset is partitioned into "K" distinct, non-overlapping clusters. Data values are assigned to clusters such that the sum of the squared distances between the data in a cluster and its respective centroid (Euclidean norm) are minimized (this is the cost function of the K-means clustering method). The cost function is shown below for reference:

$$\sum_{i=1}^{k} \sum_{x \in S_i} ||x - \mu_i||^2$$

The algorithm then computes the centroid of each cluster and repeats this process until an "optimal" centroid is determined for each cluster. The standard K-means clustering algorithm operates as follows:

1. **Input**: Data matrix "X" (which is of dimensions n x m), p the W-metric and a constant "k" [representing the number of clusters which the data points will be assignable to].
2. Initialize k centroids
3. Assign each data point to its closest center, minimizing the Euclidean norm
4. Update centroids with mean of each cluster
5. Algorithm is repeated until convergence
6. **Output**: Centroids (dimensions k x m) and labels (dimensions n x 1)

In this study, we implement a variation of the K-means clustering algorithm known as the Wasserstein K-Means clustering algorithm, which is an EM style algorithm that searches for centroids μ while minimizing the sum of the p^{th} Wasserstein distances from each data point to the barycenter of its assigned cluster (cost function of Wasserstein K-Means clustering method).

This cost function is shown below for reference:

$$\sum_{i=1}^{k} \sum_{x \in S_i} W_p(x, \mu_i)^p$$

–

Wasserstein K-means clustering is utilized in our study because detecting market regimes is a clustering problem, whereby the goal is to identify clusters of market conditions (i.e. different regimes) and assign equities part of the SP 500 to each cluster/regime. After having done this for all equities in the SP 500, we then determine the overall regime of the market, which will inform the behavior of the automated trading strategy.

1.4 Mini-batch Gradient Descent Gradient descent is an optimization algorithm which follows the negative gradient (or slope) of a loss function to the initialized value of the parameter of interest, which in our context is the Wasserstein distance, and iteratively finds the minimum of the function. However, when the sample size of the input data becomes large, gradient descent requires running on the entire dataset, which is computationally expensive. For this study, the algorithm was adjusted slightly with the addition of mini-batch (the idea is to simply divide the input data into batches of random samples and apply gradient descent to them). And as for the gradient itself, we calculate the mean gradient for the sample points in the mini-batches as well. Together, this allows mini-batch gradient descent to be computationally more efficient than standard gradient descent.

1.5 Dataset The dataset that was used for this study included the past 5 years of hourly historical prices for all U.S. equities present in the SP 500 index as of May 2022. This data was extracted from the IEX (Investor's Exchange) via the Tiingo API [1] using parallel computing. [2]

The resulting dataset was then cleaned such that stocks which had absolute log-returns exceeding 30% or insufficient data over the 5 year timeframe for which data was extracted from the exchange, were omitted entirely. Additionally, instances of missing values (NaN) within the extracted data were forward filled, and the issue of significant volatility in stock prices arising from changes in the float of individual U.S. equities (attributable to splits, secondary issuances, etc) were resolved by manually adjusting the prices of the equities in question in the following manner:

$$\text{Adjustment Factor} = \frac{\text{New Float}}{\text{Old Float}}$$

$$\text{Adjusted Price} = \frac{\text{Unadjusted Price}}{\text{Adjustment Factor}}$$

Final step for preparing data for the study included data pre-processing where the hourly price data for each day was assembled into matrices where each row represented 5-day rolling windows, offset by 1 day (resulting in 35 data points per row, given the duration of trading days being 7 hours).

[1] https://api.tiingo.com/documentation/iex

[2] API calls for multiple stock tickers from the SP 500 were staggered amongst 4-6 cores of the local machine at any time. Resulted in the data extraction process being sped up by over 65+% (4-5 minutes as opposed to 12 minutes).

2 Proposed Method

2.1 Wasserstein's Metric w/Adjusted K-means Algorithm The Wasserstein's K-means clustering algorithm operates as follows:

–

1. **Input**: Data matrix "X" (which is of dimensions n x m) and a constant "k" [representing the number of clusters which the data points will be assignable to].
2. Initialize k centroids
3. [Expectation] Assign each data point to its closest center, minimizing the sum of the pth power Wasserstein distances (from each data point to the barycenter of its assigned cluster)
4. [Maximization] Update centroids with argmin of within-cluster Wasserstein p-distance[4]
5. Algorithm is repeated until convergence
6. **Output**: Centroids (dimensions k x m) and labels (dimensions n x 1)

2.2.2 Usage of Mini-batch K-means Clustering in Wasserstein K-Means Clustering Algorithm (also see 1.4) In the full-batch K-means clustering algorithm, the entire dataset being processed must be stored in main memory for each iteration, which significantly impacts the computation speed of the algorithm as the size of the dataset increases. Therefore, given the large size of the dataset used in our study, the mini-batch K-means clustering algorithm was used (within the structure of the Wasserstein K-means clustering algorithm), and provided a significant improvement in computational performance. When using the mini-batch K-means clustering algorithm, small randomly chosen batches are taken from the main dataset and assigned to clusters depending on the previous locations of the clusters' centroids. The centroids of each cluster are then updated using gradient descent (providing an additional improvement in computational speed compared to full-batch K-means clustering algorithm). In the context of our study, the barycenters were updated with the mean gradient from the selected samples.

[4]For the default parameter value (p = 1), the updating of the centroids updates the barycenters using the MEDIAN of the sorted values. For parameter value (p = 2), the updating of the centroids updates the barycenters using the MEAN of the sorted values.

2.2.3 Usage of ADAM Optimizer The ADAM algorithm is an optimization algorithm which is an extension of stochastic gradient descent. While stochastic gradient descent maintains a single, constant learning rate, ADAM maintains a different learning rate for each parameter of the model in question. Additionally, ADAM improves the existing computational performance of gradient descent by making use of the exponential weighted average gradient (momentum algorithm) and squared gradients (RMS Propagation algorithm; however, ADAM uses the average of the second moments of the gradients [uncentered variance] as opposed to the average of the first moment of the gradients [mean] used by RMS Propagation).

2.2.4 Computational Methods of Trading Model Additional steps taken to improve the computation speed of the automated trading model designed around the Wasserstein K-means clustering algorithm included parallel computing & Bayesian optimization. Parallel computing was implemented by running the algorithm for each U.S. equity present in the SP 500 in parallel, distributing these calls amongst 4-6 cores on our local machine. In addition, Bayesian optimization, a strategy for estimating global minima when the objection function does not have a close-formed expression and thus computationally intensive to evaluate. The basic idea of this searching process is to approximate the objective function as Gaussian Process. Then we sequentially update our posterior distribution to find minima. Empirically, it shows improved performance compared with random grid search. This is more commonly seen in deep learning though.

3 Data Analysis

We will mainly have three parts. The first one is a simulation study on a synthetic price path. The second on is a trading simulation on real-life dataset. The last one is comparison with our base and optimized algorithm.

(3.1) Merton Jump Diffusion Price Paths consisting of two regimes. There are 10 regime changes in total. Please refer to the Simulation Study section.

(3.2) Trading Model on the real-life Dataset as describe in Introduction section. Please refer to the Trading Model and Model Testing section.

(3.3) Comparison between our base algorithm (using scipy.optimize.minimize) and our improved algorithm as outlined, in terms of runtime, accuracy and convergence.

3.1 Simulation Study We generated a Merton Jump Diffusion Price Paths consisting of two regimes. This is commonly seen in financial engineering, where we are interested in pricing options. It is similar to a geometric Brownian Motion in Stochastic Process, where $dS_t = \mu S_t dt + \sigma S_t dB_t$, except that there are discrete jumps.

In our simulated data, there are 10 regime changes of fixed period 0.5 year each. Therefore, we have the true labels, making it convenient to evaluate our clustering result.

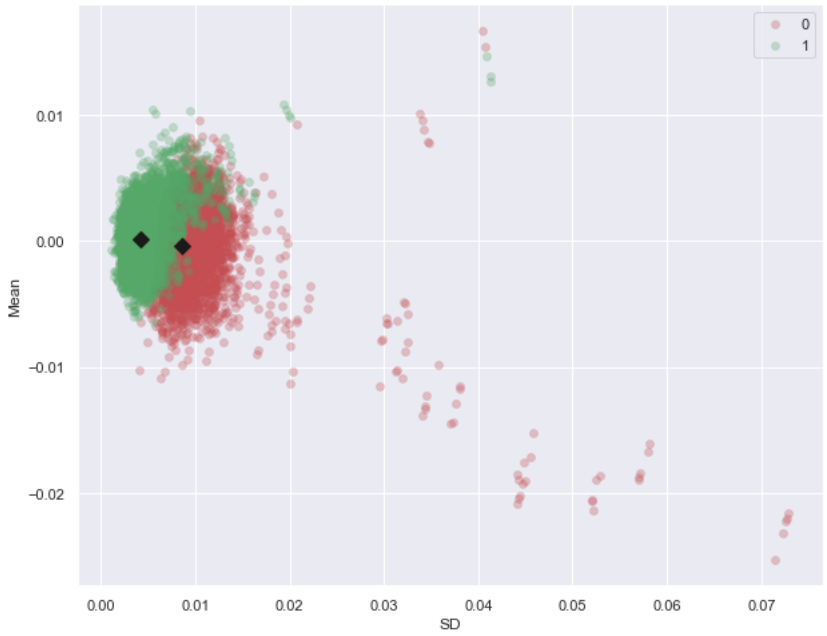


Figure 1: $\sigma - \mu$ plot of r_{lift}

We can see there is vaguely a decision boundary on standard deviation. The bearish regime has lower mean, higher volatility as expected from the data generation process.

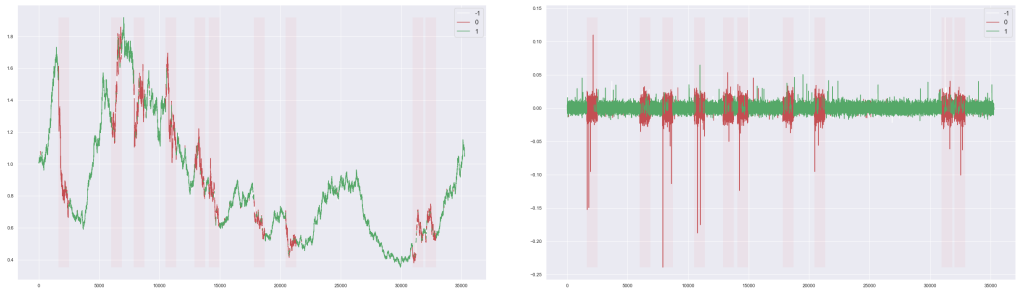


Figure 2: Price and Log-return Series from MJD model

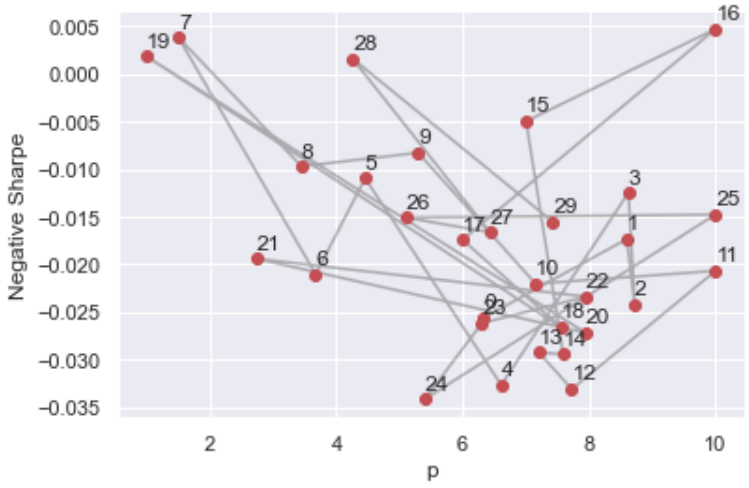
From the above two plots, where we map the labels from rolling windows back to the original time series, the red filled area is the true regime as we generated this synthetic path. As a result, we see that our Wasserstein K means almost perfectly classifies regime changes. Yet, we can still see some misclassification, such as around t = 25000, there are some red colours.

3.2 Trading Model Shown below is the trading model designed based on the Wasserstein K-mean clustering algorithm that was analyzed during this study.

The trading model executes the following process for each of the U.S. equities in the S&P 500: Within the training data for each stock (2.5 years of historical data), we first get the barrycenters from the Wasserstein K-means. Then, we have the signs of both the correlation of predicted log returns the sign of the associated barycenters' mean. If the absolute value of the correlation of predicted log returns is greater than or equal to 5%, then the model proceeds to begin trading the stock in question.

Within the validation data (1.5 years of historical data), for a particular stock that the model has begun trading on, we set weights[k] := [correlation * sign(barycenters[k].mean)]. Then, we get predicted label for each day and buy weights[k] shares. We will have the return time series using this strategy at the end of validation period. So, we can search for optimal $(p^*, k*)$ using validation Sharpe ratio := mean(return) / sd(return).

Lastly, we run the trading model using the optimal (p^*, k^*) in testing data (1 year)



We used Bayesian Optimization for hyperparameter tuning as highlighted in Section 2.2.4. We can see Bayesian optimisation cleverly samples more points that have a higher probability of being the minima.

Figure 3: PnL Chart from Validation and Test Periods

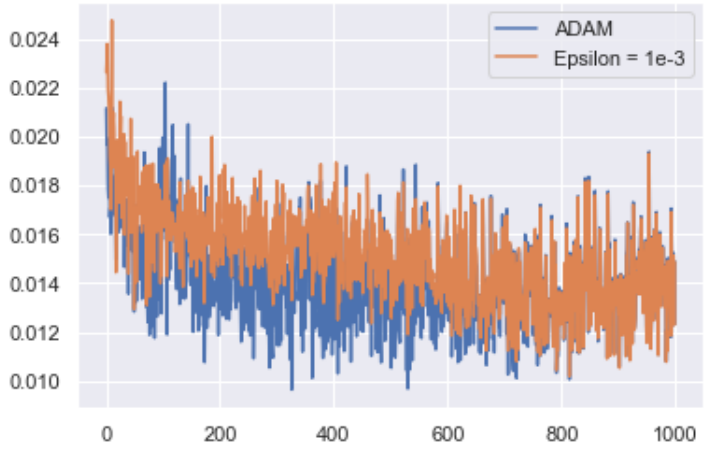
Here we give some evaluation on the usefulness and robustness of our trading signal from this Wasserstein K-Means regime prediction. The advantage is that it does achieve a higher-than-market Sharpe ratio. Besides, it outperformed the benchmark S%P500 as market was more volatile.

However, a typical optimal portfolio has a Sharpe ratio > 1. In addition, as we can see, this strategy is not market neutral. We suggest that it should be used with other signals.

3.3 Performance of Optimized Wasserstein KMeans A comparison of our improved algorithm using mini-batch against the base algorithm (which had been using full batch, and scipy.optimize.minimize for computing barycenters) is as follows in terms of runtime, accuracy and convergence.

	time	score
count	100.000000	100.000000
mean	0.749581	0.913122
std	0.177833	0.041763
min	0.640312	0.784745
25%	0.650465	0.892174
50%	0.665798	0.923065
75%	0.768111	0.942253
max	1.496578	0.975155

After running our base and optimized algorithm on the price dataset for the stock 'AAPL' (Apple Computer Inc), where the input matrix dimension is roughly 1000x35, we see that the runtime has an 100x improvement, while preserving a 90% adjusted rand score, a measure of similarity between two sets of labels.



Also, other things unchanged, we see that using ADAM optimizer converges in 300 iterations, while using constant learning rate converges in 700. That shows a more stable and faster convergence.

4 Conclusion

In conclusion, our study shows that Wasserstein's metric, when used w/the K-means clustering algorithm, is effective at identifying and distinguishing between different market regimes for large cap U.S. equities that are part of the SP 500 index. Additionally, per the trading simulation portion of our study, the regimes identified by the Wasserstein K-means can be used to design a profitable automated trading strategy. And finally, through the usage of mini-batch gradient descent, the ADAM optimizer, and parallel computing, our study has successfully demonstrated an optimized version of the Wasserstein K-means clustering algorithm that performs better than its unoptimized counterpart without altering core functionality or significantly skewing the output results (rand score ¿ 0.9 when comparing clusters formed by the optimized algorithm to those formed by the unoptimized algorithm).

It must also be noted that the trading strategy developed around the Wasserstein K-means clustering algorithm has generated a higher Sharpe ratio than the broader market over the past two years, and has outperformed the market (proxied by SP 500) during this timeframe. However, the strategy is not market neutral, and generates a Sharpe ratio that is below 1 (an optimal Sharpe ratio would be above 1), leading us to conclude that the Wasserstein metric is best used in conjunction with other signals when trading as opposed to being used as the sole signal in a trading strategy.

Taking all of this into account, the results of our study will likely cause the Wasserstein metric to be used as a signal in future regime trading strategies that focus on large cap U.S. equities. For future research into this metric, it would be beneficial to employ other clustering methods apart from K-means to test the effectiveness of the Wasserstein metric as a robust indicator of market regimes.

5 Appendix

Shown below is the code for the optimized Wasserstein K-means algorithm used in our study. Methods used: Mini-batch gradient descent, ADAM optimizer, K-Means++, parallel computing, Bayesian optimization

```
def mini_batch_wkmeans(X,p=1, k=2, state=None, batch_size=128, max_iter = 500,
a = 1e-3, b1 = 0.99,b2 = 0.999, epsilon = 1e-8,
k_means_pp = True,cal_loss = False,const_learn = False):

    """
    Technique Used: Mini Batch Gradient Descent; ADAM optimizer; Kmeans++
    Significantly reduce runtime for p != 1 and p != 2
    X: M x N matrix with M log_ret time_series
    k: Number of clusters
    p: Wp-metric, only a valid metric when p > 1
    Return: (k x N matrix that represents k centroids, array of M labels)
    """

    X_sorted = np.sort(X, axis = 1)
    rng = np.random.default_rng(state)

    if (k_means_pp):
        barycenters_0 = X_sorted[rng.choice(X.shape[0])].reshape(1,-1)

        for i in range(k-1):
            min_d_list = [distance(barycenters_0[i],X_sorted, p) for i in range(barycenters_0.shape[0])]
            mind_d = np.array(min_d_list).min(axis = 0)
            barycenters_0 = np.vstack([barycenters_0,X_sorted[min_d.argmax()]])
    else:
        barycenters_0 = X_sorted[rng.choice(X.shape[0], k, replace=False)]

    barycenters_1 = np.zeros((k, X.shape[1]))

    m = np.zeros((k,X.shape[1]))
    v = np.zeros((k,X.shape[1]))

    batch_loss = np.zeros(max_iter)
```

```
for i in range(max_iter):

    mini_batch = X_sorted[rng.choice(X.shape[0], batch_size, replace=False)]
    mini_clusters = np.array([distance(barycenters_0[c],mini_batch, p) for c in range(k)]).argmin(axis = 0)

    for j in range(k):
        if (sum(mini_clusters==j) == 0):
            barycenters_1[j] = barycenters_0[j]

        else:
            diff = mini_batch[mini_clusters==j] - barycenters_0[j]
            avg_gradient = np.multiply(-p*(abs(diff)**(p-1)), np.sign(diff)).mean(axis = 0)

            if (const_learn):
                barycenters_1[j] = barycenters_0[j] - a* avg_gradient

            else:
                m[j] = b1*m[j] + (1-b1)*avg_gradient
                v[j] = b2*v[j] + (1-b2)*avg_gradient**2
                m_hat = m[j]/(1-b1**(i+1))
                v_hat = v[j]/(1-b2**(i+1))
                barycenters_1[j] = barycenters_0[j] - a * m_hat / (v_hat **0.5+epsilon)

    if (cal_loss):
        batch_loss[i] = np.sum([distance(barycenters_1[c],mini_batch[mini_clusters == c],p).sum()
        for c in range(k)])

    barycenters_0 = barycenters_1.copy()

labels = np.array([distance(barycenters_1[i],X_sorted,p) for i in range(k)]).argmin(axis = 0)

if (cal_loss):
    return barycenters_1,labels,batch_loss

return barycenters_1,labels
```

Derivation of the gradient function:

When we update the barycenters using gradient descent, let β_{ij} be the jth coordinate of the ith barycenter.

Since our cost function is:

$$\sum_{i=1}^{k} \sum_{x \in S_i} \frac{1}{N} \sum_{j=1}^{N} |XSort_j - \beta_{ij}|^p$$

Taking partial derivative with respective to β_{ij} , we find the gradient is:

$$\sum_{x \in S_i} \frac{1}{N} p |XSort_j - \beta_{ij}|^{p-1} * sgn(\beta_{ij} - XSort_j)$$

That is why we used:

```
diff = mini_batch[mini_clusters==j] - barycenters_0[j]
avg_gradient = np.multiply(-p*(abs(diff)**(p-1)), np.sign(diff)).mean(axis = 0)
```

References

[1] Horvath, Blanka, et al. *Clustering Market Regimes Using the Wasserstein Distance* ArXiv.org, 22 Oct. 2021, https://arxiv.org/abs/2110.11848. [5]

[2] Bottou, Léon, and Yoshua Bengio. *Convergence Properties of the K-Means Algorithms - Neurips* NeurIPs, https://proceedings.neurips.cc/paper/1994/file/a1140a3d0 df1c81e24ae954d935e8926-Paper.pdf.

[3] Kingma, Diederik P., and Jimmy Ba. *Adam: A Method for Stochastic Optimization* ArXiv.org, 30 Jan. 2017, https://arxiv.org/abs/1412.6980.

[4] Merton, Robert C. *Option Prices When Underlying Stock Returns Are Discontinuous* Research Gate, Journal of Financial Economics, Jan. 1976, https://www.researchgate.net/publication/222451587_Option_Prices_When_Underlying_Stock_ Returns_Are_Discontinuous.

[5] Bergstra, James, et al. *Algorithms for Hyper-Parameter Optimization - Neurips.* NeurIPs, https://proceedings.neurips.cc/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf.

[5] In the original paper, the authors only provided the Wasserstein K-Means algorithm for p = 1 and 2. Our team replicated their results and built an actual trading model as well as derived a scalable algorithm for general p.