Barcelona School of Economics

*MSc Data Science*

MASTER THESIS

# ***Stock portfolio selection using Learning-to-rank algorithms and news sentiment***

*Authors:*

Eric Bataller

Danilo Méndez

*Supervisor:*

Prof. Dr. Argimiro Arratia

May, 2022

## I. ABSTRACT

In this thesis, we applied Learning to Rank algorithms to design strategies for trading using the performance of a group of stocks based on investors' sentiment toward these stocks. We show that learning to rank algorithms, specifically ListNet - a Neural Network algorithm for list wise ranking, is an effective and reliable way to rank the best and worst performing stocks based on investors' sentiment and other news indicators. We used different types of sentiment shock measurements in the hope of producing a ranking of the best performing stocks using Learning to Rank algorithms. We apply ListNet to 8 years worth of training data starting in 2012 to 2019 and 2 years worth of testing data from 2019 to 2021. We used long the top $\alpha\%$ of stocks as our investment strategies, based on the rankings outputted by our Neural Network. We demonstrated that the portfolio strategy applied to the ranking produced by the ListNet algorithm produces 2 year cumulative returns significantly superior to the S&P500 index return, and the buy&hold strategy of our entire stock universe (544% vs 177% vs 196%, respectively).

## II. INTRODUCTION

The impact of investors' sentiment on the financial market have been observed in behavioural finance studies [1]. Given that sentiment is the opinion of the public on the state of the financial market we need to find an adequate proxy for the investor's opinion. One of the most reliable proxies, due to their influence on investors' decisions, is news media. The arrival of news continually updates investors' understanding and knowledge of the market and influences investor sentiment as a result. In lieu of the complex nature of selecting the news media that has the best proxy of investors' sentiment, there have been some private companies that have their curated selection of news media and parse news articles to calibrate investor sentiment through text mining and machine learning techniques.

Based on such indicators, one can employ a machine learning ranking method, learning-to-rank algorithms, to construct equity portfolios based on news sentiment and other financial indicators. One of these ranking algorithms, Listwise learning-to-rank, was used recently for constructing equity portfolios, with promising results [2]. Before this, many machine learning algorithms have been mostly focused on constructing portfolios based on return forecasting. Song, Qian, Liu, Anqi and Yang proposed a stock portfolio construction approach from the viewpoint of ranking investors' relative views on stocks' performance using Listwise learning-to-rank algorithms. These are a class of algorithms that use supervised machine learning approaches to solve ranking problems, such that the model can sort new objects according to their degrees of relevance. The identification of which assets will outperform others in the future can be easily represented as a raking problem. Making an investment decision is essentially to rank all the assets in the stock universe and buy the top ranked ones or short the bottom ranked ones. Using investors' sentiment provides a natural way to rank stocks based on their reported performance in news media.

The advantage of using learning-to-rank algorithms in portfolio selection is that unlike more traditional machine learning methods that predict a value per stock, learning-to-rank provides a ranking for stocks as an output. This means that learning-to-rank targets on relative orders instead of absolute values. This property is particularly useful for investment applications. This approach presents a relative performance among stocks, and it is more reliable than an absolute performance forecast.

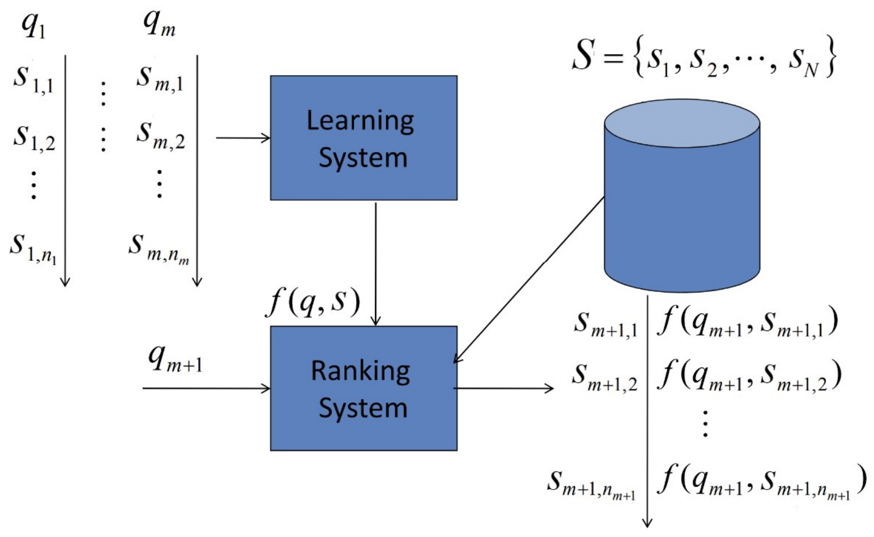
Using news indicators along with more traditional indicators bridges the gap of predicting relative performance for a group of stocks through multiple information sources. We show that Listwise learning-to-rank is effective in producing reliable ranking models to predict the best and the worst performing stocks based on news indicators and market information. In the experiments of portfolio strategies, we apply ListNet to stock selection processes and test long-only and long-short strategies. Applying back testing of the models for the period from 2012 to 2021, we show that the selected portfolios using our learning-to-rank methods significantly outperform the S&P500 index.

There are three main learning-to-rank algorithms, while we will mainly focus on the Listwise approach the other two are worth mentioning. The high level difference between these 3 approaches are the amount of elements (in our case stocks) considered in the loss function when training the model. The *pointwise* approach looks at a just one stock at a time and predicts a relevance score independent of the rest of the stocks. All the standard regression and classification machine learning algorithms can be used directly for pointwise learning-to-rank. This method isn't the most appropriate for stock ranking using news indicators since it doesn't leverage the fact that we are looking for a relative ranking of our stock universe, instead of an absolute ranking. The second method is *pairwise* learning-to-rank which looks at a pair of stocks at a time and tries to come up with the optimal ordering of the pair. Then it compares the ordering to the true ranking. The goal is to make the least amount of mistakes when comparing the pair ranking to the true ranking. This is an improvement from the pointwise learning-to-rank algorithm but it increases the amount of comparisons the algorithm must make since we would have to check every possible pair (with some practical improvement) while not looking at the bigger picture. The third approach is the approach we take in this thesis. *Listwise* learning-to-rank looks at the entire universe of stocks and tries to provide the optimal ordering (permutation). This is more complex but provides a truly relativized ordering of the stocks which is what we are looking for.

## III. LEARNING TO RANK PROBLEM FOR STOCK PORTFOLIO SELECTION

The Learning-to-Rank (LTR) setting consists of the following (see Fig. 1 for visual support). The system maintains a collection of items or stocks (in a LTR literature these are often called *documents*). Given a query $q$, the system retrieves stocks that fulfil the query specifications from the collection, ranks the items, and returns the top stocks (as well as the bottom ones if required). The ranking task is achieved by making use of a scoring model $f(q,s)$ to sort the stocks, where $s$ denotes a stock, or the desired item put to rank. In the past, the creation of the ranking model $f(q,s)$ was done without the use of any Machine Learning technique. Originally, this technique was designed for information retrieval problems such as document retrieval and collaborative filtering. In many Language Models for Information Retrieval (LMIR), for example, it is assumed that $f(q,d)$ ($d$ for document) is represented by a conditional probability distribution $P(q|d)$. One seeks to model this distribution based on the words appearing in the query and the document, and thus no training is needed (only tuning of a small number of parameters is necessary).

A new trend arose in the past decades to employ machine learning techniques to construct $f(q,d)$. The underlying idea is to give the algorithm some signals/information of relevance that might be of interest for the ranking task, and let the algorithm learn the best combination of criteria, building the mapping function $f(q,d)$ in its process. In the following subsections we describe the main issues and obstacles one must face when approaching the ranking problem via LTR techniques (with special emphasis on LTR for stock portfolio selection).



$q_1$
$s_{1,1}$
$s_{1,2}$
$s_{1,n_1}$
$q_m$
$s_{m,1}$
$s_{m,2}$
$s_{m,n_m}$
Learning System
$f(q,s)$
$q_{m+1}$
Ranking System
$S=\{s_1, s_2, \cdots, s_N\}$
$s_{m+1,1}$ $f(q_{m+1}, s_{m+1,1})$
$s_{m+1,2}$ $f(q_{m+1}, s_{m+1,2})$
$s_{m+1,n_{m+1}}$ $f(q_{m+1}, s_{m+1,n_{m+1}})$

FIG. 1: Learning to rank general setting

## A. Training and Testing

Just like any other supervised learning task, Learning to Rank requires of a training and testing stage. Denote a set of all stocks we have data on (our "stock universe") as $\mathcal{S}$, and the set of all possible queries we can use in order to retrieve stocks from our stock universe as $\mathcal{Q}$. Suppose that $\mathcal{Y} = \{1, 2, ..., l\}$ is the label set, where labels represent grades ($l \succ l-1 \succ ... \succ 1$, where $\succ$ denotes the order relation). Further suppose that $\{q_1, q_2, ..., q_m\}$ is the set of queries for training and $q_i$ is the $i$-th query. Associated with the query $q_i$, we have the following set of stocks

$S_i = \{s_{i,1}, s_{i,2}, ..., s_{i,n_i}\}$, and their corresponding set of labels $\mathbf{y_i} = \{y_{i,1}, y_{i,2}, ..., y_{i,n_i}\}$. Notice that $n_i$ denotes the size of the list of stocks $S_i$ (and its corresponding $\mathbf{y_i}$). In general, such size is going to vary for different queries. $s_{i,j}$ denotes the $j$-th stock in $S_i$; and $y_{i,j} \in Y$ denotes the $j$-th grade label in $\mathbf{y_i}$, representing the relevance degree of $s_{i,j}$ with respect to $q_i$. Our finite training data set consists of $m$ triplets $D_{train} = \{(q_i, S_i), \mathbf{y_i}\}_{i=1}^{m}$.

A feature vector $x_{i,j} = \phi(q_i, s_{i,j})$ is created from each query-stock pair $(q_i,s_{i,j})$ for $i$=1,2,...,$m$ and $j$=1,2,...,$n_i$; where $\phi$ denotes the feature functions. In other words, features are defined as functions of a query stock pair. Transferring the stock portfolio selection to a ranking problem, an example of query could be formed on each week such that the stocks associated with it fulfil some criteria (e.g amount of news reports, certain amount of trading volume, or limiting the capitalization of the companies). The group of stocks $S_i$ associated with that query (week + criteria) will have its corresponding true labels' vector $\mathbf{y_i}$ (which we take from the rankings of weekly returns), and the corresponding **list** of feature vectors $\mathbf{x_i} = \{x_{i,1}, x_{i,2}, ..., x_{i,n_i}\}$, where each feature vector $x_{i,j}$ will be computed for stock $s_{i,j} \in S_i$ based on the data available from that week (or the past). Letting $\mathbf{x_i} = \{x_{i,1}, x_{i,2}, ..., x_{i,n_i}\}$, we can now represent the training data set as $D_{train} = \{\mathbf{x_i}, \mathbf{y_i}\}_{i=1}^{m}$. Here $x \in \mathcal{X}$ and $\mathcal{X} \subseteq \mathcal{R}^d$ denotes the feature space.

Our objective here is to train our model with $D_{train}$ to come up with a mapping function that can assign a score to each query-stock pair. We have two options here: we could either aim to find a (local) ranking model $f(q,s) = f(x)$ that outputs a score to a given feature vector $x$, or we could as well consider training a global ranking model $F(q,S) = F(\mathbf{x})$ in order to obtain a function that outputs a list of scores. The local ranking model outputs a single score, while the global ranking model outputs a list of scores.

Let the stocks in $S_i$ be identified by integers $\{1, 2, ..., n_i\}$. We define a permutation $\pi_i$ on $S_i$ as the permutation chosen by our model out of the set of all possible permutations on $\Pi_i$. In that sense, we can use $\pi_i(j)$ to express the rank of the $j$-th stock (i.e $s_{i,j}$) in permutation $\pi_i \in \Pi_i$. Our ranking task is nothing but finding $\pi_i \in \Pi_i$ for a given query $q_i$ and its associated stocks $S_i$ using the scores given by our ranking model $F(q,S)$ (or $f(q,s)$), where the proper $F(q,S)$ is obtained via minimization of a *loss function*.

The test data will then consist of a new query $q_{m+1}$ and its associated stocks $S_{m+1}$. Thus, $D_{test} = \{(q_{m+1}, S_{m+1})\}$. We will create a list of feature vectors $\mathbf{x_{m+1}}$ out of the pairs in $(q_{m+1}, S_{m+1})$, and then use our ranking model (trained with $D_{train}$) to assign scores to each stock in $S_{m+1}$ and give the ranking $\pi_{m+1}$.

One may have noticed that the supervised ranking problem is somehow similar to supervised learning with respect to training and testing data but differs in that different queries and their associated stocks $(q_i, S_i)$ form a group. These groups are assumed to be i.i.d., while the intra-group data within the same group $(q_i, s_{i,j})$ are not i.i.d data. A (local) ranking model is a function of query and one stock, or equivalently, a function of feature vector $x$ derived from query and stock.

## B. Ranking Metrics

The evaluation on the performance of a ranking model is carried out by comparison between the ranking lists outputted by the model and the ranking lists given as ground truth, via a *ranking metric*. Several ranking metrics have been widely used in the literature (e.g MRR, Precision at k, MAP, DGG and NDCG). These metrics are often embedded into a loss function (designed for our model) and then we aim to minimize that loss during our training. In this study we make use of the *Normalized Discounted Cumulative Gain (NDCG)* as our ranking metric of choice to assess the quality of the result sets outputted by our model.

In order to understand NDCG, we first introduce its building blocks *Cumulative Gain (CG)* and *Discounted Cumulative Gain (DCG)*:

### *1. Cumulative Gain (CG)*

Imagine we already have our model trained to output a ranking lists $\pi_i$ when we pass a query $q_i$. Each item in the list has a ground truth relevance score $y_{i,j}$ associated with it. The higher the relevance of the $j$-th stock, the bigger $y_{i,j}$ will be. Suppose for a moment that our algorithm aims at obtaining the $k$ most important/highest performing stocks, and we don't really care about the order among the $k$ first items in the ranking. A good ranking metric for such setting would be the Cumulative Gain:

$$CG_k = \sum_{j:\pi_i(j)\leq k} y_{i,j} \tag{1}$$

The higher the $CG_k$ is, the more relevant items the list contains. Here, if our model returns a ranking $\pi_i = \{1, 2, ..., k, k+1, ...\}$, this would yield the same $CG_k$ as $\pi_i = \{k, k-1, ..., 1, k+1, ...\}$. In other words, $CG_k$ does not take into account the rank of the result set when determining the usefulness of a result set, it only cares about catching the $k$ most important items and put them at the beginning of the list, no matter the order.

Clearly, the fact that $CG$ doesn't care about the rank is not a desired behaviour one would like to have on its loss function when training an algorithm to rank items. A solution to this problem is the introduction of some sort of penalization for highly relevant documents that are put lower in the list. This is exactly what Discounted Cumulative Gain does.

### *2. Discounted Cumulative Gain (DCG)*

To overcome the aforementioned problem $CG$ has, $DCG$ penalizes highly relevant documents that appear lower in the output ranking by reducing the graded relevance logarithmically, proportional to the position. For the $k$ first items/stocks in the ranking:

$$DCG_k = \sum_{j:\pi_i(j)\leq k} \frac{y_{i,j}}{log_2(1+\pi_i(j))} \tag{2}$$

Furthermore, more emphasis can be given to the task of retrieving relevant items/stocks (without bothering about the order), by substituting the numerator by a non-linear strictly increasing function of $y_{i,j}$ such as $2^{y_{i,j}} - 1$. Thus, $DCG_k$ would look like this:

$$DCG_k = \sum_{j:\pi_i(j)\leq k} \frac{2^{y_{i,j}} - 1}{log_2(1+\pi_i(j))} \tag{3}$$

The fact is, in general, that $DCG_k$ can be thought as a combination of a *Positional discount function* $P(\pi_i(j))$ that weights less the lower positions in the ranking $\pi_i$, and a *Gain function* $G(j)$ of stock $j$ that returns the reward for the inclusion of $j$ in the top positions of the result list.

$$DCG_k = \sum_{j:\pi_i(j)\leq k} G(j)P(\pi_i(j)) \tag{4}$$

Where the functions chosen in our case happen to be: $P(\pi_i(j)) = \frac{1}{log_2(1+\pi_i(j))}$ and $G(j) = 2^{y_{i,j}} - 1$.

### *3. Normalized Discounted Cumulative Gain (NDCG)*

An issue arises with $DCG$ when we want to compare results in the ranking performance from one query to the next because the resulting lists might have different lengths ($k$ and $k'$ for example). Hence, by normalizing the $DCG$ by the maximum possible $DCG$ attained via the optimal ranking $\pi_i^*$ (obtained by sorting $\mathbf{y_i}$ in descending order), we arrive at the *Normalized Discounted Cumulative Gain* ($NDCG$). The normalizing factor is what it's known as the *Ideal Discounted Cumulative Gain (IDCG))*.

$$NDCG_k = \frac{DCG_k}{IDCG_k} \tag{5}$$

where

$$IDCG_k = \sum_{j:\pi_i^*(j)\leq k} \frac{2^{y_{i,j}} - 1}{log_2(1+\pi_i^*(j))} \tag{6}$$

Note that in a perfect ranking algorithm, the $DCG_k$ will be the same as the $IDCG_k$ producing an $NDCG_k$ of 1. All $NDCG$ calculations are then relative values on the interval $[0, 1]$ so are cross-query comparable.

### C. Formulation of LTR

We are now ready to make a formal formulation of our LTR problem as a supervised learning task. Define $\mathscr{X}$ as the input space consisting of lists of feature vectors (i.e. $\mathbf{x} \in \mathscr{X}$), and $\mathscr{Y}$ as the output space consisting of lists of scores (i.e. $\mathbf{y} \in \mathscr{Y}$). Let $P(X, Y)$ be an unknown joint probability distribution where random variable $X$ takes $\mathbf{x}$ as its value and random variable $Y$ takes $\mathbf{y}$ as its value. Assume that $F(\cdot)$ is a function mapping from a list of feature vectors $\mathbf{x}$ to a list of scores. The goal of the learning task is to learn a function $\hat{F}(\mathbf{x})$ given training data $\{(\mathbf{x_1}, \mathbf{y_1}), (\mathbf{x_2}, \mathbf{y_2}), ..., (\mathbf{x_m}, \mathbf{y_m})\}$. If wanted, $F(\mathbf{x})$ can be further written as $F(\mathbf{x}) = (f(x_1), f(x_2), ..., f(x_n))$, and $\mathbf{y} = (y_1, y_2, ..., y_n)$. Where $f(x)$ represents the local ranking function we talked about in subsection III A. In this section we will stick to $F(\cdot)$ rather than $f(\cdot)$ in order to make the formulation concise, but notice that ListNet [3], our model of choice, aims at finding a local ranking model $f(x)$. The feature vectors $x$ represent objects to be ranked (stocks, in our case).

A loss function $L(\cdot, \cdot)$ is used to evaluate the prediction result of $F(\cdot)$. First, the feature vectors/stocks in $\mathbf{x}$ are ranked according to the scores obtained by $F(\mathbf{x})$, then the top $k$ results of the ranking are evaluated using their corresponding grades $\mathbf{y}$. If the stocks with higher grades are ranked higher, then the loss function $L(F(\mathbf{x}, \mathbf{y}))$ will be small. Notice that the loss function for ranking is slightly different from the loss function from other statistical learning tasks, in the sense that it makes use of sorting. Denote the risk function $R(\cdot)$ as the expected loss function with respect to the joint distribution $P(X, Y)$ (unknown to us):

$$R(F) = \int_{\mathscr{X}\times\mathscr{Y}} L(F(\mathbf{x}), \mathbf{y}) dP(\mathbf{x}, \mathbf{y}) \tag{7}$$

Given training data, the empirical risk function would be:

$$\hat{R}(F) = \frac{1}{m}\sum_{i=1}^{m} L(F(\mathbf{x_i}), \mathbf{y_i}) \tag{8}$$

The learning task then becomes the minimization of the empirical risk function, as in other learning tasks. Minimizing it might be difficult due to the nature of the loss function (the rank metrics that are usually embedded in the loss function are not differentiable with respect to the models parameters, thus making the optimization via gradient-based methods not possible). For this reason, we consider using a surrogate loss function $L'(F(\mathbf{x}, \mathbf{y}))$ and its empirical risk:

$$\hat{R}(F) = \frac{1}{m}\sum_{i=1}^{m} L'(F(\mathbf{x_i}), \mathbf{y_i}) \tag{9}$$

Notice that we could even add a regularization term and conduct a minimization of the (regularized) empirical risk.

Ideally, the true loss function can be defined based on $NDCG$ (or other metrics such as $MAP$). For example:

$$L_{NDCG}(F(\mathbf{x}), \mathbf{y}) = -NDCG \tag{10}$$

Note that the true loss function makes use of sorting via $NDCG$, making the loss function not differentiable. A helpful toy example to see why that is, would be to think of a list of only 2 elements $(1, 2)$, where 2 has more relevance than 1. Since $NDCG$ only cares about the ranking positions, a swap between the two elements would make our ideal loss function jump unsmoothly from 0 to -1 in one step. Thus, the non differentiability of the function. We therefore need to find a proper surrogate loss that serves us as a proxy. For this, we have different options, which lead to different approaches to learning to rank (Pointwise, Pairwise, and the one we are going to focus on: Listwise).

Listwise loss functions are defined on lists of objects, just like the true loss functions, and thus are more directly related to the true loss function. Different Listwise loss functions are exploited in the Listwise methods. In the next section we explain how ListNet, our Listwise LTR algorithm of choice, approaches the ranking problem by using a probabilistic model with a tractable surrogate loss, and then, utilizes Artificial Neural Networks (NN) to solve the optimization problem.

## D. ListNet: A probabilistic model for ranking with a differentiable surrogate Loss

ListNet was firstly introduced in 2007 by Cao, Qin, Lui, Tsai and Li [3]. All of the following ideas, definitions and theorems we refer to can be found in their article with more detail. An explanation of these theorems will shed light on the methodology we will use to solve the learning problem.

We will use a probabilistic approach where we will define the permutation probability and the top one probability (explained below). We map a list of scores to a probability distribution using one of the previously mentioned probabilities and then using any metric between probability distributions as a loss function.

Following the same notation we have been using so far, suppose that there is a set of objects to be ranked and that they can be identified by integers form 1 to $n$. A permutation $\pi = \{\pi(1), \pi(2), \ldots, \pi(n)\}$ is a bijection from $\{1, \ldots, n\}$ to itself where $\pi(j)$ denotes the position in the permutation of the $j$-th object. Suppose now that there is a ranking function $(f)$ that assigns a score to each object. Let $k = (k_1, k_2, \ldots, k_n)$ be the list of scores where $k_j$ is the score of the $j$-th object. The results of the ranking function and the score list have interchangeable properties meaning that the order of the score results in the permutation that is outputted by the ranking function.

We also assume that any permutation is possible but that different permutations have different likelihood calculations based on the ranking function. We define the permutation probability in a way that it corresponds to the likelihood of a permutation given the score list.

**Definition 1 (Permutation Probability)** *Suppose that $\pi$ is a permutation of the $n$ objects and $\phi(\cdot)$ is an increasing and strictly positive function. Then, the probability of permutation $\pi$ given the list of scores $k$ is defined as:*

$$P_k(\pi) = \prod_{j=1}^{n} \frac{\phi(k_j)}{\sum\limits_{i:\pi(i) \geq \pi(j)} \phi(k_i)} \tag{11}$$

*where $k_j$ is the score of the $j$-object.*

This is the first of the permutation probabilities. We have some important properties for this definition. Given a ranking/score of all objects (given a ranking function), if we swap an object with a high score with an object of a lower score we obtain a ranking list of lower probability. This means that there is always a permutation that maximizes the likelihood of a ranking function. This also means that, according to this model, the permutation produced by sorting the ranking function is the most likely to occur. Given two lists of scores we can calculate both permutation probability distributions and use a distance metric between those as our listwise loss function. The problem here is that we need to calculate $n!$ permutations which quickly becomes unfeasible as $n$ grows. This leads us to the second definition.

**Definition 2 (Top One Probability)** *The top one probability of object $j$ is defined as*

$$P_k(j) = \sum_{\pi \in \Pi_n : \pi(j)=1} P_k(\pi) \tag{12}$$

*where $P_k(\pi)$ is the permutation probability of $\pi$ given k, and $\Pi_n$ is the space of all permutations of $\{1, 2, \ldots, n\}$.*

The top one probability of an object represents the probability of the object being ranked at the top or, in other words, have the highest score of all objects given permutation from a ranking function. It isn't clear how this definition solves our unfeasibility problem since, in order to calculate n top one probabilities, we would still need to calculate $n!$ permutations. This is solved by the following handy theorem, as it turns out that we can calculate top one probability in an efficient way, without having to go over $(n-1)!$ permutations.

**Theorem 1** *For top one probability $P_k(j)$, we have*

$$P_k(j) = \frac{\phi(k_j)}{\sum_{i=1}^{n} \phi(k_i)} \tag{13}$$

*where $k_j$ is the score of object $j$ where $j = 1, 2, \ldots, n$*

If we choose $\phi(\cdot)$ to be the $exp(\cdot)$ function we obtain:

$$P_k(j) = \frac{e^{k_j}}{\sum_{i=1}^{n} e^{k_i}} \tag{14}$$

We recognize this expression as a version of the softmax which creates a probability distribution. As we will see later, this comes very handy during the optimization process in the architecture of a Neural Net, since we only need to apply the softmax to the scoring function and thus obtain our output layer. This definition shares the same properties as the previous one. Meaning that the permutation produced by the ranking function is the most likely to occur given that $\forall i, j \in \{1, ..., n\} : i \neq j$, if $k_j > k_i$ then $P_k(j) > P_k(i)$. We use the top one probability since it is more tractable and has the same mathematical properties that we desire. With its use, given two lists of scores we can use any metric to represent the distance (listwise loss function) between the two score lists. For example, when we use Cross Entropy as metric, the listwise loss function becomes:

$$L(\mathbf{k}, \mathbf{y}) = -\sum_j \frac{exp(y_j)}{\sum_i^n exp(y_i)} \cdot \log\left(\frac{exp(k_j)}{\sum_i^n exp(k_i)}\right) \tag{15}$$

or equivalently from a minimization standpoint:

$$L(\mathbf{k}, \mathbf{y}) = -\sum_j y_j \cdot \log\left(\frac{exp(k_j)}{\sum_i^n exp(k_i)}\right) \tag{16}$$

Where $\mathbf{k}$ is the list of scores given by the model for a given query $\mathbf{k} = F(\mathbf{x}) = (f(x_1), f(x_2), ..., f(x_n))$, and $\mathbf{y}$ is the list of true labels for the stocks in that same queries, so the loss function can be written as a function of $\mathbf{x}$ and $\mathbf{y}$: $L(\mathbf{x}, \mathbf{y})$. This is known as the softmax cross entropy, and it's the loss upon which we will build our optimization problem.

Before moving on to the next section, note that the fact that we replaced our ideal loss with a proxy comes with a cost: consistency. Though a recent work [4] establishes a link between the ListNet loss function and NDCG under strict conditions (requiring binary relevance labels). In a general setting, its loss is only loosely related to ranking metrics, and it's been proven to not be NDCG-consistent [5]. Thus, when using ListNet, one can only hope to train their model with a softmax cross entropy loss, while getting closer to the optimal $L^*_{NDCG}$, always keeping in mind that such occurrence is problem dependent.

### E. ListNet: Using Neural networks to optimize the Listwise loss function

In the previous sections we have fully described our problem, its intrinsic obstacles from an optimization perspective, and a work around to partially overcome them. It is now time to decide which learning method is going to help us approximate the optimal function $f^*(x)$ that minimizes our loss $L(\mathbf{x}, \mathbf{y})$. In the case of ListNet, as its name indicates, the model proposed is a Neural Network. Specifically, a multi layer perceptron network (MLP) that learns via Gradient Descent optimization algorithm.

Our neural network aims at estimating the scores of the stocks in each query. In the rank prediction, a higher score leads to a higher rank. We denote the ranking function based on the Neural Network model $w$ as $f_w$. Given a feature vector from the i-th query $x_{i,j}$, $f_w(x_{i,j})$ assigns a score $k_{i,j}$ to it. Given query $q_i$, the ranking function $f_w$ can generate a score list $\mathbf{k}_i(f_w) = (f_w(x_{i,1}), f_w(x_{i,2}), ..., f_w(x_{i,n_i}))$. Then, the top one probability is calculated for stock $s_{i,j}$, and from there, the loss for query $q_i$ becomes:

$$L(\mathbf{k}_i(f_w), \mathbf{y_i}) = -\sum_j^{n_i} y_j \cdot \log\left(P_{\mathbf{k}_i(f_w)}(j))\right) \tag{17}$$

We can then perform backpropagation to get the gradient with respect to the parameter $w$ as follow:

$$\Delta w = \frac{\partial L(\mathbf{k}_i(f_w), \mathbf{y_i})}{\partial w} = -\sum_{j=1}^{n_i} y_{i,j} \frac{\partial f_w(x_{i,j})}{\partial w} + \frac{1}{\sum_{j=1}^{n_i} exp(f_w(x_{i,j})} \sum_{j=1}^{n_i} exp(f_w(x_{i,j})) \frac{\partial f_w(x_{i,j})}{\partial w} \tag{18}$$

Eq.(18) is then used in Gradient Descent to update the parameters of the NN:

$$w_{i+1} = w_i - \eta \times \Delta w_i \tag{19}$$

Notice that ListNet uses stocks lists as instances to update the weights and optimize the listwise loss function. An update is therefore given for each query, rather than each stock. The NN might be trying to optimize our probabilistic loss function, however we must remember that our true objective is NDCG. For this reason, as we will see again later, while training our NN we will continuously monitor NDCG via a callback in the model, in order to early-stop training after observing a lack of improvement from NDCG in the validation dataset.

## IV. FEATURE VARIABLES

### A. News Sentiment indicators

Sentiment analysis is the branch of NLP that refers to any measure by which subjective information is extracted by textual documents. The sentiment indicators that were used consist on extracting the polarity of the expressed opinion in a range spanning from positive to negative.

On a high level, Lexicon-based unsupervised sentiment classification consists of composing a dictionary of words, synthetic phrases or emoticons that denote a specific sentiment. *Good, excellent* or *terrific* might be associated with positive sentiment while *bad, horrible* or *terrible* might be associated with negative sentiment. Proportions of the amount of positive and negative words or phrases provide a measure of the strength of the sentiment in the text. Of course, there are more sophisticated variants than just using proportions, but their explanation fall out of the scope of this work.

The correlation between news sentiment and market return indicates that sentiment can be an indicator to market movements [2]. However, this time series property may not be effective on cross-sectional analysis of a group of assets. Therefore a sentiment indicator is needed for each of the companies that we are looking to rank.

Sentiment is quantified as positive and negative probabilities so that we can customize the formula for our sentiment score. The fields we used for sentiment calibration in this study are listed below.

1. Date: The date of each news article

2. Returns: The daily log return of each stock

3. - Sentiment: The predominant sentiment value for a piece of news with respect to a stock (i.e., 1 for positive sentiment and -1 for negative sentiment)

Sentiment is processed in the following way. Given a lexicon $L_\lambda$ and a stock ticker $G_k$. A sentiment score based on $L_\lambda$ for document $D_{n,t,k}$ relative to target $G_k$ is:

$$S_{n,t}(\lambda, k) = \sum_{i=1}^{I_d} w_i S_{i,n,t}(\lambda, k) \tag{20}$$

where $S_{i,n,t}(\lambda, k)$ is the sentiment value given to $i$ unigram appearing in the document and according to lexicon $L_\lambda$, being this value 0 if unigram is not in lexicon. $I_d$ is the total number of unigrams in the document $D_{n,t,k}$ and $w_i$ is a weight, for each unigram, that determines the way sentiment scores are aggregated in the document.

The indicators we used in this thesis can be broken up into two categories: sentiment and volumetric indicators. We have four sentiment indicators.

- splogsrc = $\frac{1}{2} log \left( \frac{1+positive}{1+negative} \right)$
- linsrc = $100 * \frac{positive}{positive+negative}$
- positivePartsrc = $positive$
- negativePartsrc = $negative$

The volumetric indicators are obtained in a different way.

- RVT = Volume of news of Company / Volume of news of ALL (per day)
- RCV = $100 * \frac{volume-AVG-90-volume}{\max(volume, avg-90-volume)}$

Where RVT is the relative volume and RCV computes the change in terms of volume, i.e. the relative change concerning the mean volume received over a 90-day period. Figure (2) provides a sanity check on the stationary nature of the news indicators on a randomly selected stock via a Dickey–Fuller stationarity test.

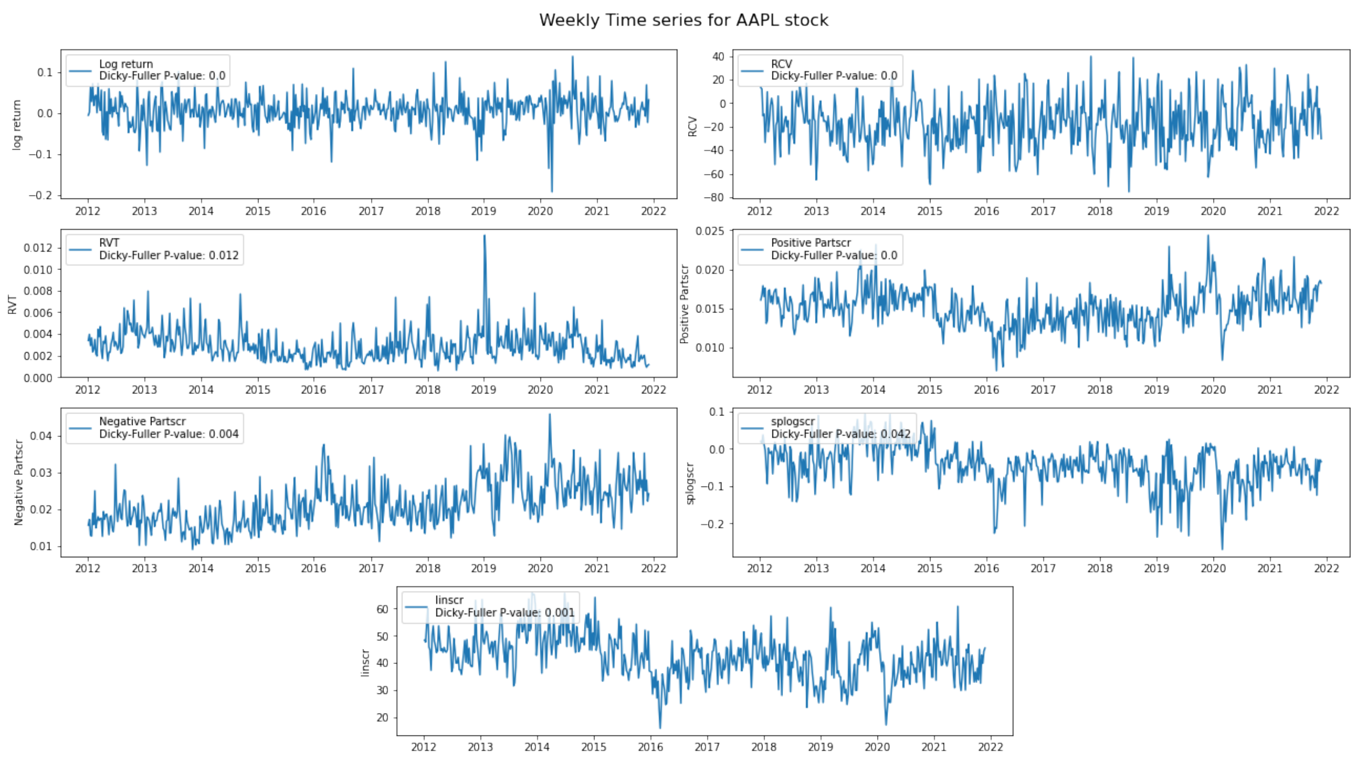


FIG. 2: Time series of all news indicators for the APPL stock

## B. Lags

Time series forecasting models require previous information to model future observations. The most simple model AR(1) only looks back one time period to predict the next. We can use the generalized version of AR model for multiple dimensions (stocks) using the vector autoregressive model (VAR). Estimating the parameters of our VAR model via OLS, we can find the coefficients for the lags and then look at the statistical significance by performing a T-test, in order to include the ones that show predictive power for the problem at hand as features.

We are assessing the lags of the log returns and seeing how significant they are as a predictor of future returns. Using VAR lag selection analysis we found that a lag of 1 period had a p value $< 1\%$, a lag of 3 had a p value $< 5\%$ and the lag for a month had a p value $< 5\%$. Providing us with 3 features that have a significant effect on the future of the log returns.

## C. Final Transformations

Apart from the transformations used in the creation of the features themselves, a final transformation was used before consolidating the dataset: Standard Scaling. Standard Scaling scales each input variable separately by subtracting the mean and dividing by the standard deviation to shift the distribution to have a mean of zero and a

| | Log Return | RCV | RVT | Positive | Negative | Splogsrc | Linsrc |
|---|---|---|---|---|---|---|---|
| Mean | 0.005 | -17.197 | 0.003 | 0.015 | 0.021 | -0.044 | 42.367 |
| Std | 0.038 | 20.382 | 0.001 | 0.002 | 0.006 | 0.056 | 8.339 |
| Min | -0.193 | -75.363 | 0.001 | 0.007 | 0.009 | -0.271 | 15.849 |
| Max | 0.137 | 39.685 | 0.013 | 0.024 | 0.046 | 0.094 | 65.807 |
| Median | 0.007 | -17.775 | 0.003 | 0.015 | 0.020 | -0.039 | 42.771 |

TABLE I: APPL stock summary statistics

standard deviation of one. We perform such transformation because, as it's been well established, Neural Networks as scale sensible. The weights of the model are initialized to small random values and updated via an optimization algorithm in response to estimates of error on the training dataset. Given the use of small weights in the model and the use of error between predictions and expected values, the scale of the inputs used to train the model are an important factor. Unscaled input variables can result in a slow or unstable learning process. In order to avoid such problem, we perform Standard Scaling on each feature of our dataset.

## V. DATA

Acuity Trading SL provided sentiment scores for each company mentioned in each news article from a curated selection of news papers for the period of January 2012 to November 2021. Market data was obtained from Yahoo Finance for the same period. We select stock from companies trading in the New York Stock and NASDAQ Exchanges.

### A. Stock Universe

We define our stock universe for portfolio selection by following these two steps:

- Select stocks with high liquidity. We obtained the stocks with the highest market capitalization in the US, and we filter them to include only the big-caps (market cap higher than \$10B).
- Filter out stocks with few news sentiment data. We excluded stocks with more than 50% missing news sentiment data.

We extract a list of 108 stocks. Our selected stock universe contain the stocks with the highest quality news sentiment data available.

### B. News indicators

According to the filtered stock universe, we group associated news articles by week. Then we select a stock to view the summary statistics (Table I) of this stock as an example.

## C. Missing data statistics

According to the filtered stock universe, we group associated news articles by week. Then we summarize statistics about the weekly news sentiment indicators. To work with a large amount of missing data we need to understand how it is distributed along each week, since we are trying to model the weekly ranking of stocks. In Figure 3 we note most stocks have at most 2 missing sentiment indicators a week throughout the year. This speaks about the wealth of weekly sentiment indicator data that we have at our disposal

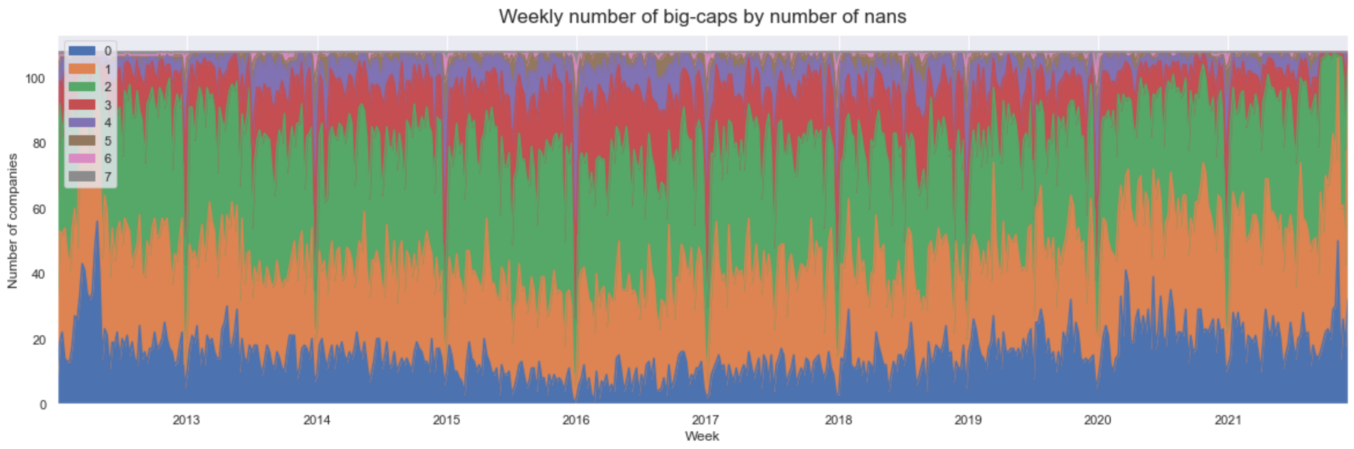


FIG. 3: Weekly number of stocks by number on NaN's

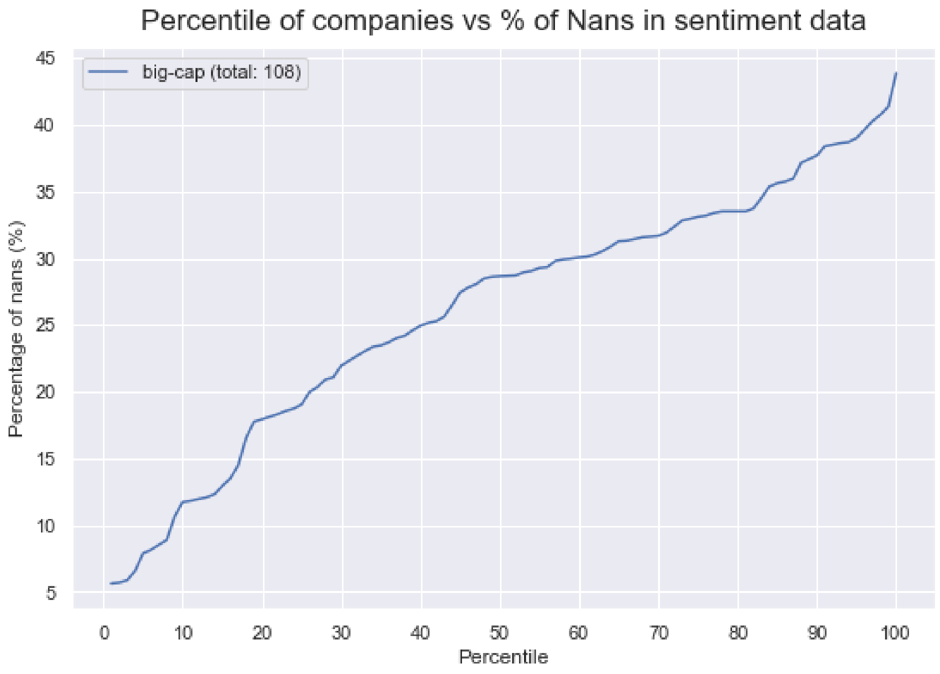


FIG. 4: Percentile of stocks vs percentage of missing sentiment data

In Figure 4 we observe the percentiles stocks vs missing sentiment data. In addition we aggregated weekly sentiment and volumetric indicators and obtained the mean to be provided as features for training.

## VI. IMPLEMENTATION AND EXPERIMENT

In order to breach the gap from theory to a practical application we lean on the TensorFlow Ranking library [6]. TF ranking is implemented on top of the Tensorflow platform, equipping it with the necessary tools to perform Learning-to-Rank (LTR). Namely, commonly used loss functions, commonly used ranking metrics, multi-item scoring functions, and other useful tools related to LTR. This allows us to use Listwise ranking algorithms and apply them to a neural network with relatively ease. In the following subsections we go over the whole process of implementation, going from the Strategy design, to the Performance Evaluation of our final model, passing through the data pipeline creation and hyperparameters tunning.

### A. Strategy design

We start by designing which kind of output we are interested in, which is ultimately related to which kind of investment strategy we want to follow. In investment strategies, we target on predicting the return rankings. The key point is to transform predicted rankings from the neural network into trading signals. In this study, we follow the rule of "long the top $\alpha\%$". Notice that we are interested in solving a portfolio selection problem, rather than a portfolio optimization problem. Thus, our investing strategy is as simple as it gets, equally weighting the stocks in our portfolio. Of course, if the ranking algorithm was able to nail the best performing stock consistently, our investment strategy would be to go long only on the first stock. Nevertheless, since there's only so much our algorithm can predict (as we will see in the following sections), a smarter strategy would probably be to go long on the top $\alpha\%$ of the stock universe, where $\alpha$ will be decided by the investors willingness to take on risk.

An important design decision is to choose how often we will update our portfolio of stocks. In that sense, in order to preserve the flexibility of being able to change our portfolio quickly without compromising too much the predictive power of our sentiment indicators, it makes sense to use a weekly investment strategy. The query will therefore be formed out of the stocks available on each week. An additional constraints is added onto the query formation: we tolerate at most two missing sentiment data per week so that our summary statistics are more representative of the whole week, preserving the highest quality data available. Therefore at each time period (week) we aggregate the daily sentiment indicators by taking the mean, forming the weekly sentiment features in the process.

As stated before, in section IV, we have 6 news indicators per stock, a 1 period lagged log return, a 4 period lagged log return and a 1 lagged monthly log return in each query. This represents the features that the neural network will provide a score to rank. These need to be fed grouped by query, and by stock.

## B. Data pipeline

One of the key steps when dealing with listwise ranking problems and Neural Networks is the data ingestion pipeline. Data must be fed to the NN in the appropiate form so that the network can perform the feedforward pass through all the stocks in one query, while saving the output scores in memory, only to update the weights at the end of each list of stocks. The data structure used in order to feed the model is not important, as long as the Tensorflow model receives tensors in a compatible format. In our case, the way we achieved this was to structure our data into a Protobuffer format. Protobuffers are extensible structures suitable for storing data in a serialized format, either locally or in a distributed manner [7]. TF ranking has a couple of pre-defined protobuffers such as ExampleListWithContext (ELWC) which make it easier to integrate and formalize data ingestion into the ranking pipeline. On a high level, ELWC form a dictionary of dictionaries, containing the information related to the query features in the *context* section, while the features uniquely related to each stock for that query are stored in an *example* section. Each ELWC eventually contains the information of a single query and the list of feature vectors of each stock. The ELWC records are later stored all together in a TFrecords format, which can be easily read and converted to tensors when creating a data pipeline for ranking algorithms. ELWC TFrecords make the code and the data more conducive to experimentation and more flexible to be integrated and used with various existing pipelines.

## C. Train vs Test split

We split our data into training and testing. It is important to note that: while a ListNet algorithm can be trained on a random split under general settings, in this case we want to preserve the temporal dimension intact so that we can effectively backtest our strategy later on a continuous period to see how the algorithm would perform under normal conditions. Thus, we do a 70vs30 split, where the training period corresponds to the 364 weeks on the interval [2012/01/08 - 2018/12/09] and out testing period corresponds to the 156 weeks on the interval [2018/12/10 - 2021/12/05]. Unfortunatly, given that our dataset is relatively scarce, with only 520 weeks, we decided to use our our testing data both for testing and validation of the neural net. This might obviously impose a bias on the resulting model, but we rather do that than compromising our training dataset with a further split.

## D. Model creation and Hyperparameters tuning

We decided to use the Keras API to build our Neural LTR model as we believed it would help us prototype, iterate and experiment changes in our model faster in a notebook environment. As it's been already mentioned, our estimator will be a MLP that we will train on the Softmax Cross Entropy loss function. However, since our true objective is $NDCG$, we will continuously monitor it via a callback in the model, in order to early-stop the training after observing a lack of improvement from $NDCG$ in the validation dataset. This way we ensure we stop the training when the validation $NDCG$ and the training loss start to diverge significantly. In particular, we set

$NDCG_{27}$ as our main target. 27 constitutes the 25% of our stock universe, and will be the base investment strategy, holding the most diversified portfolio out of the ones we will backtest.

Relatively simple architectures were proposed for our NN given that we only have 9 features: 1 to 2 layers, relatively small number of units per layers (in the range from 5 to 20 per layer), and finally a small dropout in between layers were tested during the hyperparameters tuning phase. Hyperband Tuner [8] was used as our hyperparameters search model which, in short, consists of a Bandit-Based Approach to optimization where predefined resources like iterations, data samples, or features are allocated to randomly sampled configurations. This usually provides a speedup over brute force competitors such as Grid Search. After several iterations, the winning architecture seems to be a Neural Network with batch size of 16, 1 hidden layer, 10 hidden nodes, relu activation function, a learning rate of $\mu = 0.5$, and a dropout rate of 0.05.

## E. Performance evaluation

The evaluation of the model's performance is done was done by the $NDCG_{27}$ metric on our selected model. Fig. (5) shows the saturation of both train and validation $NDCG_{27}$ after snot so many epochs. Notice that the oscillations on the training curve are most likely due to the dropout regularization. Approximately around 100 epochs, the model arrives at its maximum performance (around 0.69 $NDCG_{27}$).

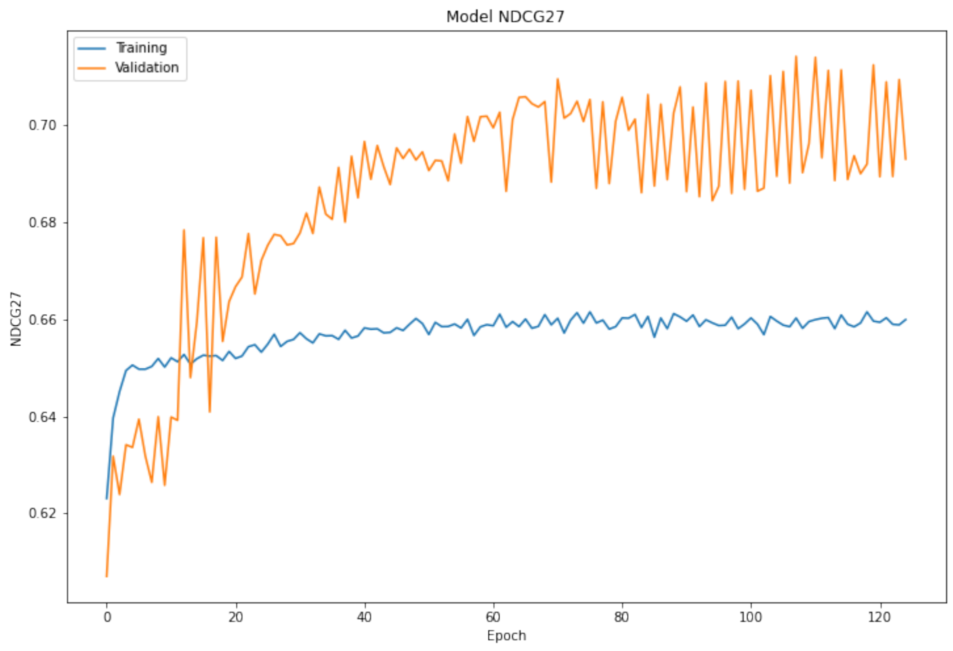


FIG. 5: Training and validation evaluation metric

## VII. RESULTS - BACKTESTING

After obtaining the optimal model, it's time to assess what the true performance of the algorithm would be, had we applied our portfolio selection method in a market environment. We run experiments of long-only strategies using ListNet algorithm for different levels of $\alpha$% (or equivalently, different portfolio sizes). The backtest period is from 2018/12/10 to 2021/12/05, which covers a high volatility regime during the 2019 COVID pandemic and the posterior economic recovery period. To justify the performance of our strategies, we choose the "Buy&Hold strategy" of the S&P500 index and our Stock universe as benchmarks for comparison. A detailed comparison of the performance between strategies can be found in Table II, containing the main metrics of investment strategy evaluation. Namely, Cumulative return, Compounded Annualized Growth Rate (CAGR), Sharpe Ratio, Max Drawdown (Max DD), and Annualized volatility. Additionally, the evolution of the Cumulative return through the backtesting period can be seen in Fig.(6).

As it can be seen, all the strategies put to test outperform the benchmarks according to their cumulative returns. In particular, ListNet performs progressively better as we reduce the size of our portfolio, showing the effectiveness of the model when it comes to predicting the future rankings. Notice that: as we keep increasing the portfolio size, the capacity of the model to correctly assess the relevance of the high ranking positions diminishes, which makes the investment strategies with portfolio size 27,15 and 10, not that different between each other. Despite that, they still achieve higher returns than the benchmarks.

The value of all strategies drop in the bear market as it can be seen by the Maximum Drawdown metric, but our LTR strategies seem to be less robust against sudden market downturns, probably because they are more exposed due to smaller diversification and the fact that they are forced to go long. However, the LTR strategies perform notably better both during the economic recovery period and the period prior to the 2019 COVID pandemic.

Surprisingly enough, the investment strategy with best score in terms of Sharpe Ratio, is our LTR long 2%, with a 0.99, proving the consistency of the algorithm in achieving high returns. The strategy is followed by the "Stock Universe B&H" and "S&P500 B&H", which score better than the other LTR strategies. Nevertheless, further testing should be applied, as one would expect the Sharpe Ratio of the smallest portfolio strategy to be smaller due to the reduced diversification and the risk associated with it.

## VIII. CONCLUSION

In this thesis, we created a stock portfolio selection approach utilizing learning-to-rank algorithms on news sentiment indicators to capture relative performance of stocks. Listnet, a learning-to-rank algorithm, was used

| | S&P500 B&H | Stock Universe B&H | LTR 2% | LTR 5% | LTR 9% | LTR 14% | LTR 25% |
|---|---|---|---|---|---|---|---|
| Cum. Return | 1.77 | 1.96 | 5.44 | 3.04 | 2.27 | 2.18 | 2.17 |
| CAGR | 0.20 | 0.25 | 0.72 | 0.43 | 0.30 | 0.28 | 0.28 |
| Sharpe Ratio | 0.96 | 0.98 | 0.99 | 0.80 | 0.63 | 0.69 | 0.81 |
| Max DD | 0.32 | 0.36 | 0.61 | 0.57 | 0.53 | 0.47 | 0.44 |
| Ann. Volatility | 0.21 | 0.25 | 0.72 | 0.53 | 0.46 | 0.40 | 0.35 |

TABLE II: "S&P500 Buy&Hold" investment strategy against "Stock Universe Buy&Hold" and "LTR long $\alpha$%"

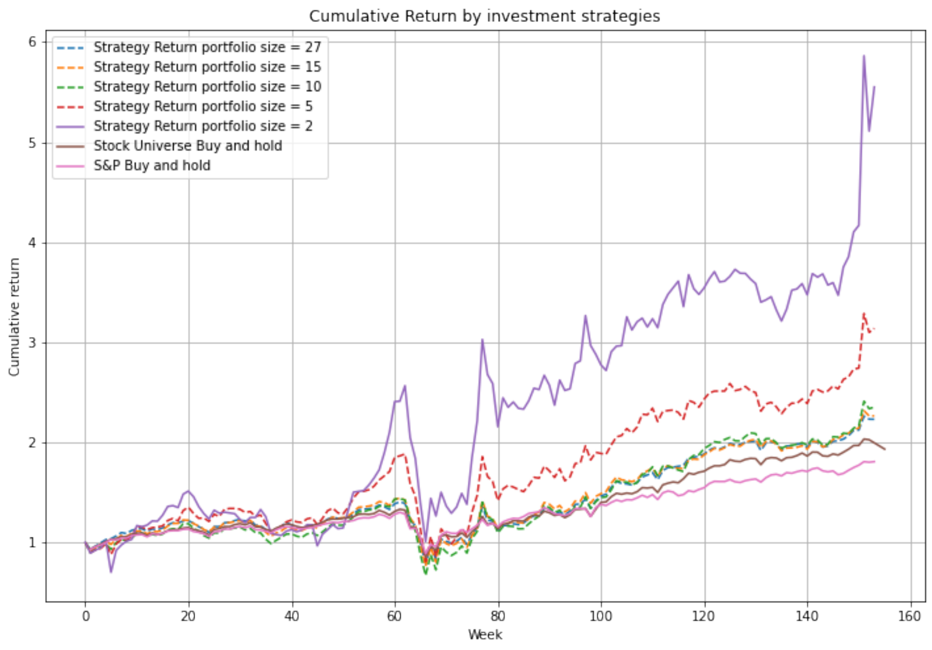


FIG. 6: Cumulative return curve by strategy - comparison across Backtesting period

in the experiments of portfolio strategies. Our data includes features, such as historical market returns, news sentiment indicators and lagged returns. A curated selection of stocks was made using stocks that had at most 50% missing sentiment data. Of those, we selected the top 107 with the highest market capitalization and trained on the ranking provided naturally by the log returns for the period of 2012-2019 and tested on the period of 2019-2021.

The backtesting results indicate that the portfolio selected by Listnet significantly outperformed the S&P500 index (544% vs 177% Cumulative return). The superior performance is consistent under different market conditions. However, the portfolio strategies put to test in the experiments seem to be more sensible to sudden market downturn, probably due to the smaller degree of diversification and their small degree of flexibility (only long positions strategies were put to test). Notice however, that this study's objective was to provide a consistent method for portfolio selection, rather than portfolio optimization. In this regard, ListNet proved to offer really good results with few data, being able to rank stocks consistently and effectively.

The overall merit of the proposed trading strategies is that these strategies are based on investors' relative views manifested through news sentiment toward the individual stocks' future performance. However, one limitation of our work is that the rebalance frequency of the trading strategies is weekly; further optimization can be done to balance the trading cost and profit.

## IX. DISCUSSION

It isn't immediately clear why a shorter selection of stocks provides a more consistent portfolio than a less volatile portfolio such as the top 27 stocks. Learning-to-rank algorithms provide a ranking that is relativized, which provides no information as to where the returns turn from positive to negative. The top 27 stocks at each time period may have contained negative returns, this is specially relevant with a bearish market. Using only the top 5 stocks at each time period provides better results since our model is able to rank efficiently. Betting on the winning horses, we are able to reduce the number of stocks with negative log returns from our selection. With less stocks one would expect more volatility, which is true for our case, but we also note that the risk is not as high as expected. Interestingly enough, we note that when the S&P500 index dipped during the COVID-19 pandemic, our selection of stocks also dips but it stays to the same levels as the S&P500 index does, or even higher. The high Maximum Drawdown scores indicate an underlying threat to the robustness of our model: When the overall economy goes bad in a shock, the fact that our investment strategy is forced to go long constraints our ability to be flexible enough to avoid the dips. However, this is partially a portfolio optimization problem, and not only a portfolio selection one.

It is important to note that it is specially hard to tune a ranking model with such few data. We used 107 stocks and over 500 queries to train the data, leading to over 500 thousand data entries. This might seem a lot, but notice that in Listwise learning-to-rank algorithms, the Neural Network parameters only get updated during back propagation. Meaning that we effectively have only around 500 possible updates per epoch. More data can help tune the Neural Network better but the cost of creating historical news indicators may be prohibitive. Nonetheless, results are promising considering that we achieved great results with such low volume of data.

An impact of the features on the model performance can be found by creating models with different features and comparing their performance with the same trading strategies. This would create a more robust model selection process than just hyper-parameter tuning and selecting the model with the highest NDCG.

## X. FUTURE STEPS

### A. Weight Composition

One problem with the output that learning-to-rank algorithms provide is that the weight composition of a portfolio isn't clear. A potential solution to this problem is to include volatility in our response variable. That is, in this thesis we only used the log-returns as our response variable, we can instead use a risk-adjusted metric that includes volatility and the investor's tolerance for risk using the formula:

$$\max_{\mathbf{w}} L(\mathbf{w}) \text{ where, } L(\mathbf{w}) = \lambda \mathbf{w} \mu - (1-\lambda)\mathbf{w}\mathcal{C}\mathbf{w}^T$$

subject to certain constraints where $\mathbf{w} = (w_1, \dots, w_k)$ with $w_i$ is the proportion of the total investment allotted to stock $i$. $\mu = (\mu_1, \dots, \mu_k)$ is the expected return of stock $k$ and $\mathcal{C}$ is the covariance matrix of all possible stocks. This is known as the Markowitz's Portfolio Selection Problem. Another possible solution, and presumably a better one would be to apply this problem to the results of the original learning-to-rank algorithm to choose the distribution of weights in the portfolio. In other words, leave the Portfolio selection problem to the LTR algorithm, and perform portfolio optimization based on the predicted rankings and the historical volatility of each stock.

### B. Trading strategies

In this thesis we only considered long-only trading strategies. A long-short trading strategy would probably have provided better returns during bearish streaks, at the expense of lower overall returns.

The choice in the number of stocks at each time period is fixed, this could also be tuned. Flexible choice of number of stocks paired with long-short trading strategies could potentially increase returns.

[1] M. Baker and J. Wurgler, “Investor sentiment in the stock market,” *Journal of Economic Perspectives*, vol. 21, pp. 129–152, June 2007.

[2] Q. Song, A. Liu, and S. Y. Yang, “Stock portfolio selection using learning-to-rank algorithms with news sentiment,” *Neurocomputing*, vol. 264, pp. 20–28, 2017. Machine learning in finance.

[3] Z. Cao, T. Qin, T.-Y. Liu, M.-F. Tsai, and H. Li, “Learning to rank: From pairwise approach to listwise approach,” *Proceedings of the 24th International Conference on Machine Learning*, vol. 227, pp. 129–136, 01 2007.

[4] S. Bruch, X. Wang, M. Bendersky, and M. Najork, “An analysis of the softmax cross entropy loss for learning-to-rank with binary relevance,” in *Proceedings of the 2019 ACM SIGIR International Conference on the Theory of Information Retrieval (ICTIR 2019)*, pp. 75–78, 2019.

[5] P. Ravikumar, A. Tewari, and E. Yang, “On ndcg consistency of listwise ranking methods,” in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* (G. Gordon, D. Dunson, and M. Dudík, eds.), vol. 15 of *Proceedings of Machine Learning Research*, (Fort Lauderdale, FL, USA), pp. 618–626, PMLR, 11–13 Apr 2011.

[6] TensorFlow, “Tensorflow ranking.” `https://github.com/tensorflow/ranking`, 2019. GitHub.

[7] G. Inc, “Protocol buffers - google’s data interchange format.” https://github.com/protocolbuffers/protobuf, 2008.

[8] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, “Hyperband: A novel bandit-based approach to hyperparameter optimization,” *Journal of Machine Learning Research*, vol. 18, no. 185, pp. 1–52, 2018.