Table of Contents

Introduction

Objective

The objective of this project is to *classify* the buy signals generated by the MACD (Moving Average Convergence Divergence) indicator as correct and incorrect at the moment of generation. A trained model will output either 0, indicating a correct buy signal, or 1, indicating an incorrect buy signal, hence performing binary classification.

Labeling *(see Appendix A)*

The first green MACD Histogram bar that comes after a red MACD Histogram bar is tagged as a buy signal and the vice versa as a sell signal. If the close price of a sell candle is greater than the close price of its buy candle; then that buy signal is labeled as correct, otherwise it is labeled as incorrect. Obviously, the number of data points in the "incorrect buy signal" class will be greater than the number of data points in the "correct buy signal" class and this situation will cause a class imbalance. In order to handle this issue; methods such as under-sampling, over-sampling and class-weights are implemented.

Dataset *(see Appendix B)*

The dataset used in this project is the candlestick data provided by Binance which contains information regarding the open, high, low, close prices and the volumes of candles of certain timestamps. Binance provides candlestick data for every interval enumeration (e.g. 5 minutes, 30 minutes, 4 hours) of every pair (e.g. Ethereum/Usdt, Bitcoin/Usdt) which allows the formation of many different combinations.

Depending on the time frame, 150 to 6000 signals are generated for each pair of coins with either a neglectable or a significant class imbalance. Currently; 32 different features, most of which having a correlation coefficient lower than 0.5, are in use. Some features are complex enough to require their own repositories, however most of them are generated by using the open, high, low, close, volume values of the signal candle and the candles before the signal candle. As a result, the features are far from being independent and not all that linearly separable. Furthermore, financial data is very noisy and it is possible to encounter cases in which one of the two signals having a cosine similarity index close to 1 being labeled as correct and the other one being labeled as incorrect.

Both categorical and continuous features for the same metric can be generated from the candlestick data. For example, the relationship between the Bollinger bands and the close price can be continuously represented using the formula $\%B = \frac{Close\ Price - Lower\ Band}{Upper\ Band}$, or the close price being in between, below or above the Bollinger bands can be categorically represented by using the function $f(\%B)\begin{cases} -1\ if\ \%B < 0 \\ 0\ if\ 0 < \%B < 1 \\ 1\ if\ \%B > 1 \end{cases}$ which might have a greater contribution to the outcome from time to time.

Limitations

First of all, financial market data is a time-series data, however the information regarding the previous candles can only be provided using the feature set. By using recurrent neural networks, the information regarding the previous candles can be provided to a model more effectively. However, native recurrent neural networks face vanishing & exploding gradient problem with financial data which enforces the use of the LSTM and Transformer architectures. Considering time constraints of this project, attempting to implement an LSTM or a Transformer network might result in the possible failure of this course.

Secondly, most of the features generated and used in this project are kept confidential, but the required information regarding the feature set will be shared tacitly as needed.

Algorithms

Logistic Regression *(see Appendix C)*

Logistic Regression is a linear, statistical model used for classification that outputs a probability. The reasonings behind the choice of this algorithm includes it being efficient to train, flexible and producing interpretable model coefficients (weights). On the other hand; the main disadvantage of Logistic Regression in this project was it being a linear model, meaning that it does not perform well on non-linearly-separable data such as financial data.

In order to transform the feature set to a dimension where they would be more linearly separable, many transformation equations and their combinations were tested. In the end, the best combination of transformations obtained is as follows:

```
def RBF(self, X, gamma=None):
    if gamma == None:
        gamma = 1.0/X.shape[1]

    K = numpy.exp(-gamma * numpy.sum((X - X[:,numpy.newaxis])**2, axis=-1))

    return K

def POLY(self, X, degree=2):
    original_axis = X.shape[1]
    for d in range(2, degree+1):
        X = numpy.hstack((X, (X[:, :original_axis]**d)))

    return X

def TRIG(self, X):
    sin = numpy.sin(X)
    cos = numpy.cos(X)

    X = numpy.hstack((X, sin, cos))

    return X

transformed_features = POLY( TRIG( RBF(original_features) ) )
```

In this transformation process, the features are first passed into a radial basis function, where the gamma value is equal to $\frac{1}{\#\ of\ features}$, which means that the number of features is now equal to the number of data points. Then the *sine* and *cosines* values of transformed features are taken and added to the feature set, tripling the number of current features. The intuition behind this layer is the fact that financial data resembles sine and cosine equations, furthermore it is also known that sine and cosine functions are used in the decomposition processes of such time-series data. Finally, the square the obtained transformed features are taken and added to the feature set, doubling the number of current features.

In order to evaluate the difference between the model's performance on transformed and non-transformed features, macro F1 scores and confusion matrices are used. The reason why macro F1 score is preferred over accuracy measure is because both the training and test data had class imbalance. Furthermore; since the goal behind this model is to be able to predict if a MACD Histogram's buy signal is correct or incorrect, evaluating the model on past data is usually very misleading. Moreover; validation methods such as cross-validation are also quite specious since it does not make sense to validate a model on a data point when the data points that comes right after it are used during training. That is why the train-validation-test splits are performed in a time-series manner where the model is trained using past data and tested on recent candles. **Figure 1** and **Table 1** shows the difference in the transformed and non-transformed data's performance using macro F1 score and confusion matrices.

[1]

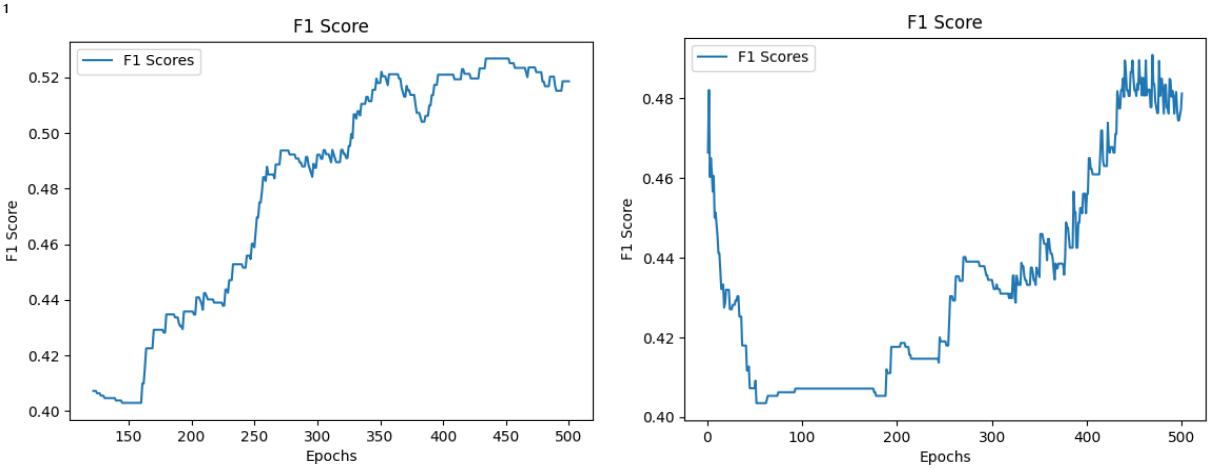

Figure 1: *Performance of Transformed Features (Left), Raw Features (Right) on validation dataset L2 Regularization and Mini-Batch Gradient Descent is used*

	Transformed Features	
	Prediction	
Ground	Correct Signal	Incorrect Signal
Correct Signal	32	49
Incorrect Signal	30	111

	Raw Features	
	Prediction	
Ground	Correct Signal	Incorrect Signal
Correct Signal	10	71
Incorrect Signal	21	120

Table 1: *Confusion Matrices generated on test dataset L2 Regularization and Mini-Batch Gradient Descent is used*

The implementation of Logistic Regression in this project aims to minimize the following loss function:

$$L(W) = -\frac{1}{M}\sum_{i=1}^{M} y_i \times \log\left(\frac{1}{1 + e^{W_i \times X_i}}\right) + (1 - y_i) \times \log\left(1 - \frac{1}{1 + e^{W_i \times X_i}}\right)$$

$$\frac{\partial L}{\partial W} = \frac{1}{M}\sum_{i=1}^{M} X_i \times \left(\frac{1}{1 + e^{W_i \times X_i}} - y_i\right)$$

However; with this loss function, the validation loss remained almost the same (with a little increase) as the training loss decreased as it can be seen in **Figure 2**. Furthermore, the model did not even produce a single "correct signal" label which means that the model was definitely not learning.

[1] see *Appendix D* for the implementation of macro F1 score

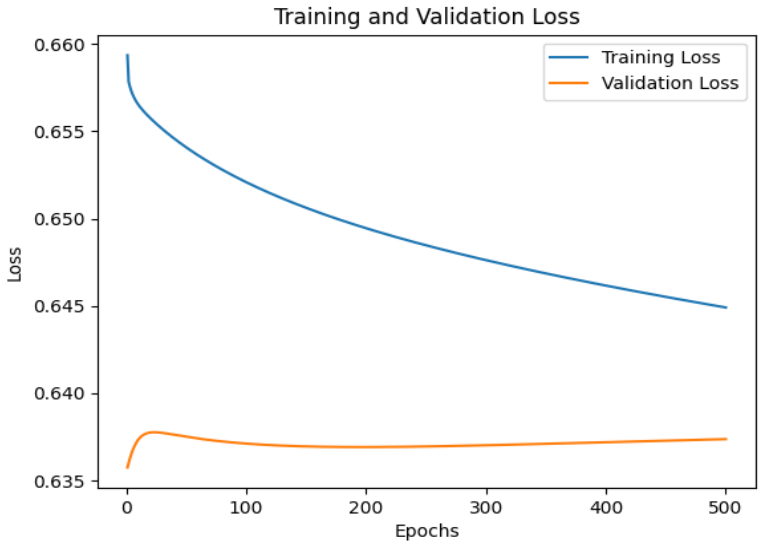

Figure 2: *Training and Validation Loss at each epoch without regularization*

In order to resolve this issue, *L2* regularization is introduced to the model which transformed the loss function to:

$$L(W) = -\frac{1}{M}\sum_{i=1}^{M}\left[y_i \times \log\left(\frac{1}{1+\ e^{W_i \times X_i}}\right) + (1-y_i) \times \log\left(1-\frac{1}{1+\ e^{W_i \times X_i}}\right)\right] + \frac{\lambda}{2M}\sum_{i=1}^{M} {W_i}^2$$

$$\frac{\partial L}{\partial W} = \frac{1}{M}\sum_{i=1}^{M}\left[X_i \times \left(\frac{1}{1+\ e^{W_i \times X_i}} - \ y_i\right)\right] - \frac{\lambda}{M}\sum_{i=1}^{M} W_i$$

The introduction of *L2* regularization drastically improved the performance of the model as the model is unable to learn the data without a significant λ value with the current hyperparameter configuration. However, *L2* regularization was also not powerful enough to make the model learn before mini-batch gradient descend was introduced to the model. **Figure 1** and **Table 1** shows the performance of the model after the addition when *L2* regularization and mini-batch gradient descend into the system. As the model was unable to output any meaningful predictions, no performance metric could even be visualized before the introduction of *L2* regularization and mini-batch gradient descend.

Apart from mini-batch gradient descend, stochastic gradient descend algorithm is also implemented. As it can be seen in **Figure 3** and **Table 2**, even though stochastic gradient descend algorithm seems to perform a little bit better than the mini-batch gradient descend on the validation dataset, it is not stable enough to be practical and it has a lesser accuracy on the test dataset compared to mini-batch gradient descend in this project. Even though it takes more iterations for stochastic gradient descend to catch up with mini-batch gradient descend, it can be stated that stochastic gradient descend algorithm trains the model faster than the mini-batch gradient descend algorithm.

	Prediction	
Ground	Correct Signal	Incorrect Signal
Correct Signal	36	4
Incorrect Signal	48	103

Table 2: *Confusion Matrix generated on test dataset with Stochastic Gradient Descend*

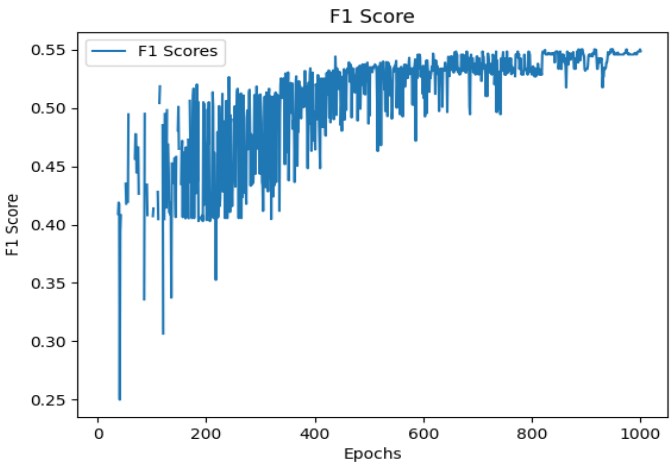

Figure 3: *Macro F1 Score generated on validation dataset at each epoch with Stochastic Gradient Descent*

The final improvement introduced to Logistic Regression in this project is class weights. Together with the addition of class weights, the loss function becomes:

$$L(W) = -\frac{1}{M}\sum_{i=1}^{M}\left[\varpi_0\left(y_i \log\left(\frac{1}{1+\ e^{W_i \times X_i}}\right)\right) + \varpi_1\left((1-y_i)\log\left(1-\frac{1}{1+\ e^{W_i \times X_i}}\right)\right)\right] + \frac{\lambda}{2M}\sum_{i=1}^{M} {W_i}^2$$

$$\frac{\partial L}{\partial W} = \frac{1}{M}\sum_{i=1}^{M}\left[\varpi_0\left(y_i\left(\frac{1}{1+\ e^{W_i \times X_i}} - 1\right)\right) + \varpi_1\left(\left(\frac{1}{1+\ e^{W_i \times X_i}}\right)(1-y_i)\right)\right] - \frac{\lambda}{M}\sum_{i=1}^{M} W_i$$

In the overall data used throughout this report, the ratio between incorrect signal labels and correct signal labels is *(1.8)*. As it can be seen in **Figure 4** and **Table 3**, the addition of class weights did not result in a significant improvement at this ratio.

	Prediction	
Ground	*Correct Signal*	*Incorrect Signal*
Correct Signal	24	57
Incorrect Signal	20	121

Table 3: *Confusion Matrix generated on test dataset with class weights*

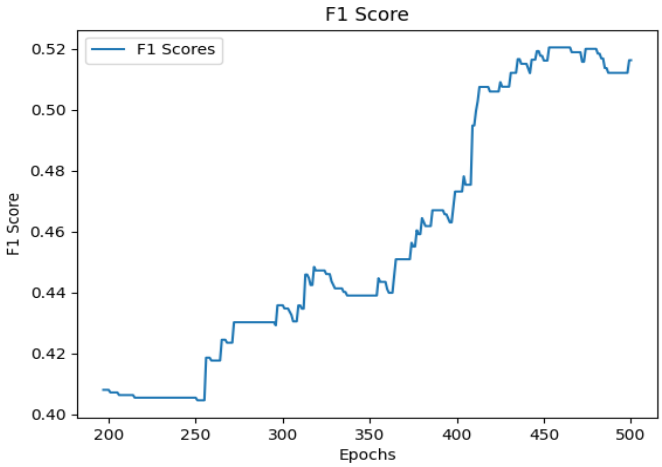

Figure 4: *Macro F1 Score generated on validation dataset at each epoch with class weights*

In the implementation of Logistic Regression in this project, the greatest macro F1 scores and best confusion matrices are always obtained at states of overfitting and underfitting. **Figure 5** shows the validation loss graph of a model, that uses mini-batch gradient descend, which produced a macro F1 score of 0.597 (the greatest macro F1 score recorded in this project). It can be seen that the validation loss first increases for a short amount of time, then it decreases for some time and finally it keeps increasing till the end of the training; implying an overfit. **Figure 6** on the other hand shows the training and validation loss graph of a model, that uses stochastic gradient descend, which produced a macro F1 score of 0.588 (the second greatest macro F1 score recorded) and it can be seen that, even though the validation loss follows a slightly downwards trend, training loss follows a horizontal trend; implying an underfit. It is important to note that even though loss and F1 score are inversely correlated in most datasets, they do not necessarily indicate the same thing.

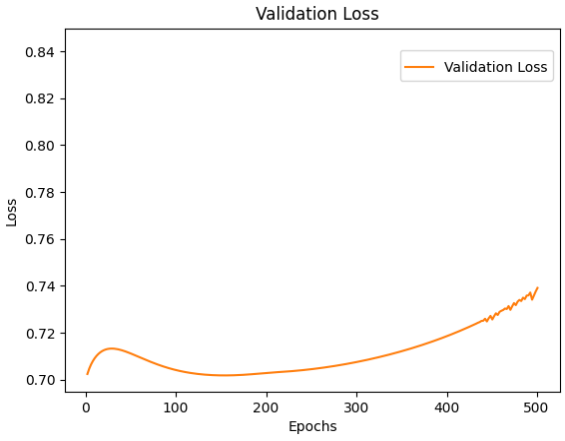

Figure 5: *Validation Loss obtained with transformed features, L2 regularization and mini-batch gradient descend*

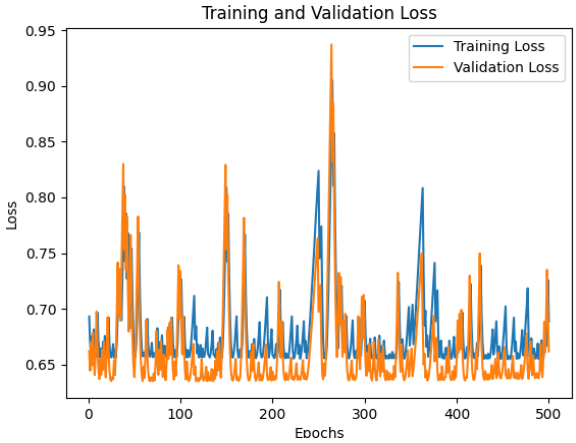

Figure 6: *Training and Validation Loss obtained with transformed features, L2 regularization and stochastic gradient descend*

It is important to note that even though loss and F1 score are inversely correlated in most datasets, they do not necessarily indicate the same thing. In this project's situation; it can be stated that as the model gets less confident on its predictions, it starts to get more accurate. It is believed that this phenomenon is justifiable as finance data is extremely noisy and it is not very much generalizable, which means that it might really be possible that overfitting the noise in the past data may results in greater accuracies in the predictions made for the recent data.

K Nearest Neighbors *(see Appendix E)*

K Nearest Neighbors is a non-linear algorithm that uses a data points proximity to other data points in order to classify the data point. This algorithm is chosen because it is similar to the K-Means algorithm that was taught in this course, it is easy to implement, time-efficient and considering the relationship between loss and macro f1 score in the case of logistic regression, it was believed that a model that uses similarity measure in order to make a prediction could be effective on the dataset used in this project.

When using K Nearest Neighbors algorithm, it is very important that all features are brought to the same scale. In this project, standardization is used in order to scale the features. However; since finance data is a time-series data, it is very essential for future data points have absolutely no effect on the previous data points in order to avoid deceptive results. That is why; instead of using the mean and standard deviation of the whole dataset to standardize a data point, the mean and standard deviation of the

previous N data points is made used of to standardize a data point in a rolling window manner[2]. Although this technique is not as effective as default standardization at bringing all features to the same scale, it still allows data points to be brought to a similar enough scale so that most of the statistical learning algorithms can produce meaningful results while avoiding producing deceptive ones. Finally; in order to prevent outlier candles from throwing the model off the track, a limit (4 and -4 by default) is applied to the maximum and minimum values that the z-score can get. This process introduces two new hyperparameters to the system which are standardization window (N) and z-score limit.

Figures 7, 8 and 9 shows the trend that macro F1 score follows for K values ranging from 1 to 20 at different standardization windows where the similarity measure is Euclidean Distance. It can be seen that even though the predictions get more stable as the standardization window increase, the greatest F1 scores are obtained when the standardization window is relatively large and when K value is between 1 and 3; which ones again indicates that overfitting the model to some extent results in more accurate predictions with less confidence.

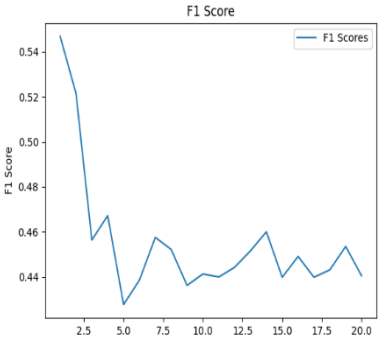

Figure 7: *Standardization window is 240 candles*

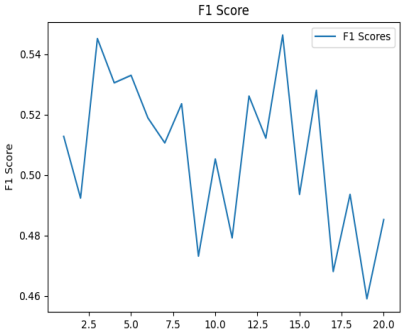

Figure 8: *Standardization window is 960 candles*

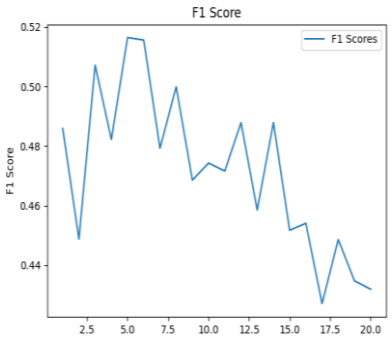

Figure 9: *Standardization window is 1680 candles*

On the other hand; when different similarity measures are used, better results are obtained without any overfitting patterns being observed. As it can be seen on the **Tables 4, 5 and 6**; cosine similarity measure results in the greatest accuracy when used by the K Nearest Neighbors algorithm. From these results; it can be deduced that, the orientation of the vector formed by the features is more important than its magnitude. This situation may also suggest that the ratio between the features is more significant than the difference between the features which would be a reasonable suggestion since profits are realized based on the percentage difference between prices. In the case of the difference between the Euclidean Distance and Manhattan Distance; it can be stated that the reason why Manhattan Distance performs better than Euclidean Distance might be the fact that Manhattan Distance is less sensitive towards outliers that "commonly" occur in financial datasets.

$$Euclidean\ Distance = \sqrt{\left(\sum_i X_i - \hat{X}_i\right)^2}$$

$$Cosine\ Similarity = \frac{\vec{X} \cdot \vec{\hat{X}}}{\|X\| \times \left\|\hat{X}\right\|}$$

$$Manhattan\ Distance = \sum_i \left|X_i - \hat{X}_i\right|$$

[2] see *Appendix F* for the implementation

Euclidean Distance

	Prediction	
Ground	*Correct Signal*	*Incorrect Signal*
Correct Signal	32	48
Incorrect Signal	44	97

F1 Score: 54.4%

Table 4: *Confusion Matrix generated by the KNN algorithm where the standardization window is 960 candles and K = 3*

Cosine Similarity

	Prediction	
Ground	*Correct Signal*	*Incorrect Signal*
Correct Signal	38	43
Incorrect Signal	38	106

F1 Score: 60.3%

Table 5: *Confusion Matrix generated by the KNN algorithm where the standardization window is 120 candles and K = 10*

Manhattan Distance

	Prediction	
Ground	*Correct Signal*	*Incorrect Signal*
Correct Signal	27	54
Incorrect Signal	28	116

F1 Score: 56.7%

Table 6: *Confusion Matrix generated by the KNN algorithm where the standardization window is 120 candles and K = 14*

Naïve Bayes Classifier *(see Appendix G)*

Naïve Bayes Classifier is another supervised learning algorithm that uses a probabilistic approach to statistical learning. Naïve Bayes Classifier is chosen as the third algorithm of this project because it works well with categorical data and it is within the context of the course. The main problem that Naïve Bayes Classifier will face in this project's financial dataset is the fact that almost all features are generated by using a small set of other features (mainly the features of open, high, low, close and volume), making them far from being independent, although the model is going to assume that all of the features are independent from each other. In order to deal with this issue, only features that gives information about a different component of the financial market (e.g., volatility, momentum, trend and etc.) will be selected.

All of the features that were given to the Logistic Regression and K Nearest Neighbors algorithms were numeric; however for Naïve Bayes algorithm, categorical features are also going to be used. As it is mentioned in the introduction (on $\%B$ indicator), certain continuous features can be intuitively converted to categorical features. Some continuous features can also be categorized using the thresholds commonly used in the literature. For example; RSI values lesser than 30 are considered to be low (indicating an oversold condition) and RSI values greater than 70 are considered to be high (indicating an overbought condition). Using such thresholds, certain features can also be categorized. Remaining features are going to be categorized by segmenting (bucketing) the data into bins after normalization.

At first, Gaussian Naïve Bayes algorithm is used on the same continuous features that were inputted to the previous algorithms (Logistic Regression and K Nearest Neighbors). In order to calculate the probabilities of a signal being correct and incorrect ($P(C_{correct} \mid x)$ and $P(C_{incorrect} \mid x)$), the following equation is used:

$$P(C_i \mid x) = \frac{P(x \mid C_i) \times P(C_i)}{P(x)}$$

$$P(C_i \mid x) \approx P(C_i) \times \prod_{j=0}^{n} P(x_j \mid C_i)$$

$$P(C_i \mid x) \approx P(C_i) \times \prod_{j=0}^{n} \frac{1}{\sqrt{2\pi\sigma_{ij}{}^2}} \times e^{-\frac{(x_j-\mu_{ij})^2}{2\sigma_{ij}{}^2}}$$

where the priors $P(C_i)$) and the means & variances of every feature (μ_{ij}, σ_{ij}) for every class are calculated at the fit function. Later on, the log-likelihood formula is implemented in order to avoid overflow and underflow problems which turned to equation into:

$$\log P(C_i) + \sum_{j=0}^{n} \log \frac{1}{\sqrt{2\pi\sigma_{ij}{}^2}} \times e^{-\frac{(x_j-\mu_{ij})^2}{2\sigma_{ij}{}^2}}$$

When the same features inputted to the previous algorithms are given to the Gaussian Naïve Bayes algorithm without any modifications, the model obviously failed to learn the data as it can clearly be seen in **Table 7**. Even though almost all of the features used follows a gaussian distribution (**see Figure 10**) as the number of samples is relatively large and standardization (z-score) is used to normalize the data, the results presented in **Table 7** was expected. The reason for this expectation was the fact that the features used are far from being independent and the use of continuous features together with the gaussian distribution amplifies the negative effects of features being dependent.

	Prediction	
Ground	*Correct Signal*	*Incorrect Signal*
Correct Signal	76	5
Incorrect Signal	129	14

F1 Score: 35.2%

Table 7: *Confusion Matrix and Macro F1 Score generated by the Gaussian Naïve Bayes*

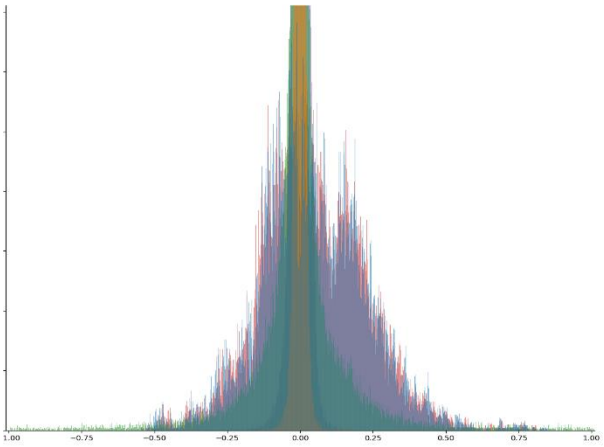

Figure 10: *Distribution plot of the features used.*

Afterwards, Categorical Naïve Bayes algorithm is used together with categorized features. Firstly, the whole categorized feature set is inputted to the Categorical Naïve Bayes model the same way continuous features were inputted to Gaussian Naïve Bayes model and a significant boost in classification performance is achieved as it can be seen in **Table 8**.

	Prediction	
Ground	*Correct Signal*	*Incorrect Signal*
Correct Signal	29	52
Incorrect Signal	46	95

F1 Score: 51.5%

Table 8: *Confusion Matrix and Macro F1 Score generated by the Categorical Naïve Bayes model (whole feature dataset)*

Then; in order to reduce the negative effects of the features that are highly dependent to each other, some features are eliminated from the feature set. The reason such feature engineering was needed is the fact that Naïve Bayes algorithms assumes features are independent from each other and using highly dependent features together may mislead the model as those features would be giving the "same" information multiple times. The elimination strategy used revolves around equalizing the number of features that give information on the market's momentum, volatility, trend and volume. Before the elimination, features that would give information on market's trend were more dominant especially due to the fact that the same feature would be used multiple times with different time periods. Even though the idea of using the same feature with different time periods performed well on Logistic Regression and K Nearest Neighbors algorithms, as it can be seen in **Table 9**, the elimination strategy used increased the accuracy of the Categorical Naïve Bayes considerably.

	Prediction	
Ground	*Correct Signal*	*Incorrect Signal*
Correct Signal	31	49
Incorrect Signal	43	96

F1 Score: 53.9%

Table 9: *Confusion Matrix and Macro F1 Score generated by the Categorical Naïve Bayes model (set of selected features)*

Evaluation & Conclusion

Performances of the Implemented Models on the Datasets from the Literature[3]

Since financial data is very complex, chaotic and mostly unpredictable; the expected macro f1-scores of the models before their implementations were in between 0.5 and 0.6. In the end, this expectation has been satisfied by the models. However; in order to get a better understanding of the quality of the models' implementations, their accuracies on the datasets from the literature needs to be observed. The breast cancer dataset from "scikit-learn" library is chosen for this purpose as it is a highly popular binary classification dataset. As it can be seen in the **Tables 10, 11 and 12**; the implemented models perform almost perfectly on the dataset judging by the high F1 Scores and properly formed confusion matrices.

Logistic Regression:

	Prediction	
Ground	*0*	*1*
0	12	2
1	0	43

Macro F1 Score: 95.0%

Table 10: *Logistic Regression's accuracy on breast cancer dataset with L2 Regularization (λ=1) and mini-batch gradient descent (# batches = 32)*

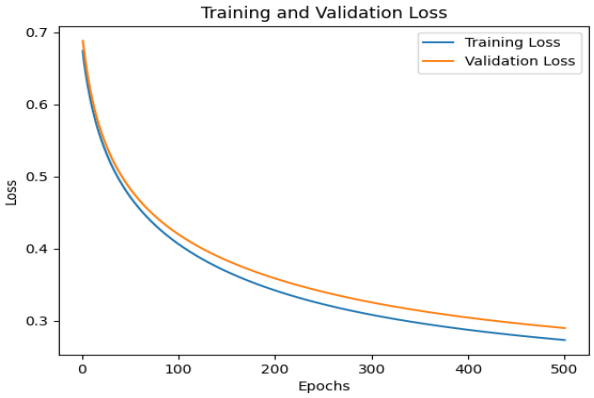

Figure 11: *Loss graph obtained during the training of the Logistic Regression model in* ***Table 10***

K Nearest Neighbors (Euclidean Distance):

	Prediction	
Ground	*0*	*1*
0	13	1
1	0	43

Macro F1 Score: 97.5%

Table 11: *Confusion Matrix and Macro F1 Score generated by K Nearest Neighbors model where K = 9 and Euclidean Distance is the similarity measure*

Gaussian Naïve Bayes:

	Prediction	
Ground	*0*	*1*
0	12	2
1	2	42

Macro F1 Score: 92.7%

Table 12: *Confusion Matrix and Macro F1 Score generated by the Gaussian Naïve Bayes model*

[3] *See Appendix H, I, J for the implementations*

Which Algorithms Performed the Best?

Considering the discussions and the data provided in the previous sections, it can be stated that Naïve Bayes Classifier had the worst overall performance out of the three algorithms tested in this project. However, this situation is no surprise since the Naïve Bayes Classifier assumes that features are independent from each other and the features in the feature set used in this project are far from being independent as almost all features are extracted from a smaller subset of other features. In other words, Naïve Bayes Classifier outputted the worst results amongst the other two algorithms as expected.

The greatest results obtained from the Logistic Regression (59.7% macro F1-score) and K Nearest Neighbors (60.3% macro F1-score) algorithms are very similar to each other. In theory, it can be stated that K Nearest Neighbors algorithm is more suited for the dataset used in this project since the feature set is not linearly separable. However, the results obtained from both algorithms being very close to each other indicates that the kernel functions used were very successful at transforming the feature set into a linearly superable hyperplane.

Nonetheless; considering the fact that the Logistic Regression model implemented have more hyperparameters which are computationally more expensive and more time consuming to optimize compared to the two simple hyperparameters of the K Nearest Neighbors algorithm, which are *K* and similarity measure, it is safe to state that K Nearest Neighbors algorithm performed the best on the dataset used in this project.

Appendix

The code repository of this report can be found through the link:
dem0-0/Statistical_Learning_Project (github.com)

Appendix A

Labeling algorithm:

```
import pandas

def macd_labeling(
      input_df: pandas.DataFrame,
      fast_period: int,
      slow_period: int,
      signal_period: int
) -> pandas.Series:
    """Label Macd Histogram's buy signals as correct & incorrect

    Args:
        input_df (pandas.DataFrame): A dataframe that contains ohlcv data.
                                     The column that holds the close price
                                     data must be named as "close" and the
                                     rows must be in ascending order by
                                     timestamp
        fast_period (int): macd fast period
        slow_period (int): macd slow period
        signal_period (int): macd signal period

    Returns:
        pandas.Series: A pandas series of length same as the length of the
                       input candles_df,
                       Correct Macd Histogram buy signals are labeled as 1
                       Incorrect Macd Histogram buy signals are labeled 2.
                       Everything else is labeled as 0
    """
    def macd_histogram(candles_df: pandas.DataFrame) -> pandas.Series:
        fast_ema = candles_df["close"].ewm(
                       span=fast_period, adjust=False,
                       min_periods=fast_period
        ).mean()
        slow_ema = candles_df["close"].ewm(
                       span=slow_period, adjust=False,
                       min_periods=slow_period
        ).mean()

        macd = fast_ema - slow_ema
        macd_signal = macd.ewm(
                       span=signal_period, adjust=False,
                       min_periods=signal_period
        ).mean()

        return macd - macd_signal

    # Copy the input dataframe to avoid modifying it
    candles_df = input_df.copy()

    # Calculate Macd Histogram
    macd_hist = macd_histogram(candles_df)
```

```
    candles_df = candles_df.assign(macd_hist=macd_hist)

    # Loc buy & sell points
    buy_sell_signals = macd_hist[macd_hist * macd_hist.shift(1) < 0]

    # If first signal is a sell signal, make it gone
    buy_sell_signals = buy_sell_signals if buy_sell_signals.iloc[0] > 0
                                        else buy_sell_signals.iloc[1:]

    # Buy sell points with ohla data
    bsp_with_ohlc = candles_df.loc[buy_sell_signals.index]

    # Correct & Incorrect signals's indicies
    correct_indicies = bsp_with_ohlc.loc[
                       bsp_with_ohlc.shift(-1).close > bsp_with_ohlc.close
    ].loc[bsp_with_ohlc.macd_hist > 0].index
    incorrect_indicies = bsp_with_ohlc.loc[
                       bsp_with_ohlc.shift(-1).close <= bsp_with_ohlc.close
    ].loc[bsp_with_ohlc.macd_hist > 0].index

    # Label the input
    candles_df["label"] = 0
    candles_df.loc[correct_indicies, "label"] = 1
    candles_df.loc[incorrect_indicies, "label"] = 2

    return candles_df["label"]
```

Appendix B

Data Collection Script:

```
import requests
import pathlib
import shutil
import pandas
import os

def _download_historical_candles(
      path: pathlib.Path,
      symbol: str,
      interval: str
):
    """Download the kline data for the given interval of the given symbol
       and merge the data into a single csv file

    Args:
        interval (str): Interval ENUM
        symbol (str): Symbol ENUM

    Raises:
        Exception: If there is a server-side error at Binance endpoint
        Exception: If no data is found to be downloaded, the parameters
                   given may be the cause
    """
    os.makedirs(path, exist_ok=True)

    base_url = f"https://data.binance.vision/data/spot/monthly/klines/\
{symbol}/{interval}"
```

```
    # For every month of every year in which kline data exists
    years = range(2017, 2024)
    months = range(1, 13)
    for year in years:
        for month in months:
            url = f"{base_url}/{symbol}-{interval}-{year}-\
{str(month).zfill(2)}.zip"
            response = requests.get(url, stream=True)

            # If the data exists, unpack the data
            if response.status_code == 200:
                zip_path = path.joinpath(f"{year}-\
{str(month).zfill(2)}.zip")
                with open(zip_path, "wb") as f:
                    f.write(response.content)

                shutil.unpack_archive(zip_path, path)
                os.remove(zip_path)
            elif int(response.status_code % 100) == 4:
                print(f"URL: '{base_url}/{symbol}-{interval}-{year}-\
{str(month).zfill(2)}.zip' does not exists")
            elif int(response.status_code % 100) == 5:
                error_message = f"{response.status_code}: Binance Server\
Error. Try again later"
                print(error_message)
                raise Exception(error_message)

    if len(os.listdir(path=path)) == 0:
        error_message = f"Something went wrong. 'symbol' or 'interval'\
parameters might not be correct"
        Logger.error(error_message)
        raise Exception(error_message)

    # Merge the data into one csv file
    columns=["timestamp", "open", "high", "low", "close", "volume"]
    final_df = pandas.DataFrame(columns=columns)
    for csv_path in sorted(os.listdir(path=path)):
        current_file = pandas.read_csv(
                path.joinpath(csv_path),
                header=None,
                names=columns+["ct", "qav", "not", "tbbav", "tbqav", "ig"])
        final_df = pandas.concat([final_df, current_file[columns]], axis=0)

    [os.remove(path.joinpath(csv_path_)) for csv_path_ in
     os.listdir(path=path)]

    # Save the final csv
    final_df.sort_values(
       by="timestamp",
       ignore_index=True
    ).to_csv(path.joinpath('ohlcv.csv'), index=False)
```

Appendix C

Logistic Regression Implementation:

```
import numpy

class LogisticRegression():
    def __init__(
      self,
      num_features: int,
      regularization: bool=False,
      constant: int=1,
      stochastic: bool=False,
      class_weights: dict=None
   ):
        self.weights = numpy.zeros(num_features+1)
        self.regularization = regularization
        self.constant = constant
        self.stochastic = stochastic
        self.cw = class_weights

    def _sigmoid(self, X):
        return 1 / (1 + numpy.exp(-X))

    def _log_loss(self, pred, ground):
        loss = numpy.sum(ground * numpy.log(pred) + (1 - ground) *
                                        numpy.log(1 - pred)) / -len(ground)

        if self.regularization == True:
            loss += ((self.constant / len(ground) / 2) *
                                            (self.weights.T @ self.weights))

        return loss

    def fit(self, X: numpy.ndarray, y: numpy.ndarray, lr: float):
        # Add bias feature
        X = numpy.hstack((numpy.ones((X.shape[0], 1)), X))

        # Get model predictions
        linear_pred = numpy.dot(X, self.weights)
        logistic_pred = self._sigmoid(linear_pred)

        # Calculate the gradient of the weights
        if self.stochastic == True:
            # If stachastic gradient descend is active
            rand = numpy.random.randint(low=0, high=X.shape[0]-1)
            if self.cw is None:
                # If class weights are active
                dWeights = X[rand] * (logistic_pred[rand]- y[rand])
            else:
                dWeights = X[rand] *
                           ((self.cw[0]*y[rand]*(logistic_pred[rand]-1)) +
                           (self.cw[1]*logistic_pred[rand]*(1-y[rand])))
        else:
            # If stachastic gradient descend is deactive
            if self.cw is None:
                # If class weights are active
                dWeights = (1 / X.shape[0]) * (X.T @ (logistic_pred - y))
```

```
            else:
                dWeights = (1 / X.shape[0]) *
                           (X.T @ ((self.cw[0]*y*(logistic_pred-1)) +
                           (self.cw[1]*logistic_pred*(1-y))))

        if self.regularization == True:
            # If regularization is applied
            dWeights -= (1 / X.shape[0]) * (self.constant * self.weights)

        # Update weights
        self.weights -= lr * dWeights

        # Calculate & return loss
        loss = self._log_loss(logistic_pred, y)
        return loss

    def predict(self,
                X: numpy.ndarray,
                threshold: int,
                y: numpy.ndarray=None
    ):
        # Add bias feature
        X = numpy.hstack((numpy.ones((X.shape[0], 1)), X))

        # Get model predictions
        linear_pred = numpy.dot(X, self.weights)
        logistic_pred = self._sigmoid(linear_pred)

        # Match model predictions with classses
        pred_classes = [1 if pred >= threshold else 0 for pred in
                                                            logistic_pred]

        # Calculate the loss of the predictions
        if y is not None:
            pred_loss = self._log_loss(logistic_pred, y)
            return pred_classes, pred_loss

        return pred_classes
```

Appendix D

Implementation of Macro F1 Score:

```
import numpy

def f1_macro(ground, pred):
    labels = numpy.unique(ground)
    f1 = 0
    for label in labels:
        true_positive = numpy.sum((ground==label) & (pred==label))
        false_positive = numpy.sum((ground!=label) & (pred==label))
        false_negative = numpy.sum((pred!=label) & (ground==label))

        precision = true_positive/(true_positive+false_positive)
        recall = true_positive/(true_positive+false_negative)

        f1 += 2 * (precision * recall) / (precision + recall)

    return f1 / len(labels)
```

Appendix E

Implementation of K Nearest Neighbors:

```
import numpy

class KNearestNeighbors():
    def __init__(self, k: int):
        self.k = k

        self.X_train: numpy.ndarray
        self.y_train: numpy.ndarray

        self.similarity_measure: str

    def fit(self, X: numpy.ndarray, y: numpy.ndarray):
        self.X_train = X
        self.y_train = y

    def _predict(self, x: numpy.ndarray):
        if self.similarity_measure == "euclidean":
            distances = [
                  numpy.sqrt(numpy.sum(x-x_train)**2)
                  for x_train in self.X_train
            ]
        elif self.similarity_measure == "cosine":
            distances = [
                  (x @ x_train.T)
                  /
                  (numpy.linalg.norm(x) * numpy.linalg.norm(x_train))
                  for x_train in self.X_train
            ]
        elif self.similarity_measure == "manhattan":
            distances = [
                  numpy.sum(abs(x-x_train)) for x_train in self.X_train
            ]

        if self.similarity_measure == "cosine":
            # A greater cosine similarity value means a lower distance
            k_indices = numpy.argsort(distances)[::-1][:self.k]
        else:
            k_indices = numpy.argsort(distances)[:self.k]

        k_nearest_labels = [self.y_train[i] for i in k_indices]

        most_common_label = max(
                        set(k_nearest_labels), key=k_nearest_labels.count
        )

        return most_common_label

    def predict(self, X: numpy.ndarray, similarity_measure: str):
        if similarity_measure not in ["euclidean", "cosine", "manhattan"]:
            raise Exception("The 'similarity_measure' parameter must be \
either one of: ['euclidean', 'cosine', 'manhattan']")
        self.similarity_measure = similarity_measure

        predictions = [self._predict(x) for x in X]
        return predictions
```

Appendix F

Rolling standardization function:

```
import pandas

def rolling_zscore(
      column: pandas.Series,
      window: int,
      limit: int
) -> pandas.Series:
    """Apply zscore normalization on the given column by looking at the
       past values to calculate the mean and standard deviation in order to
       avoid looking to the future.
    Args:
        column (pandas.Series): A pandas series of numeric values
        window (int): Size of the rolling window
        limit (int): Maximum value that the absolute value of zscore can
                     be.Used for dealing with outliers
    Returns:
        pandas.Series: Standardized column
    """
    rolling_mean = column.rolling(window=window).mean()
    rolling_std = column.rolling(window=window).mean()

    rolling_zscores = (column - rolling_mean) / rolling_std

    rolling_zscores[rolling_zscores > limit] = limit
    rolling_zscores[rolling_zscores < -limit] = -limit

    return rolling_zscores
```

Appendix G

Implementation of Naïve Bayes Classifier:

```
import numpy

class NaiveBayes():
    def __init__(self, method: str="gaussian"):
        if method not in ["gaussian", "categorical"]:
            raise Exception("Only Gaussian & Categorical Naive Bayes \
algorithms are supported. The 'method' parameter must be either gaussian \
or categorical."
        )
        self.method = method

        self._classes: list

        self._prior: numpy.ndarray

        if method == "gaussian":
            self._mean: numpy.ndarray
            self._var: numpy.ndarray
        elif method == "categorical":
            self._probs: list[list[dict]] = []

    def fit(self, X: numpy.ndarray, y: numpy.ndarray):
        n_samples, n_features = X.shape

        self._classes = numpy.unique(y)

        # Initialize prior
        self._prior = numpy.zeros(len(self._classes), dtype=numpy.float64)

        # If Gaussian Naive Bayes is used, initialize mean and variance
        if self.method == "gaussian":
            self._mean = numpy.zeros(
                                (len(self._classes), n_features),
                                dtype=numpy.float64
            )
            self._var = numpy.zeros(
                                (len(self._classes), n_features),
                                dtype=numpy.float64
            )
        elif self.method == "categorical":
            self._probs = [0] * len(self._classes)

        for index, label in enumerate(self._classes):
            X_label = X[y == label]

            # Calculate the prior for each class
            self._prior[index] = len(X_label) / n_samples

            # If Gaussian Naive Bayes is used, calculate mean and variance
            if self.method == "gaussian":
                self._mean[index, :] = X_label.mean(axis=0)
                self._var[index, :] = X_label.var(axis=0)
```

```
            elif self.method == "categorical":
                feature_probs = [
                    {
                        category: sum(X_label[:, feature_idx] == category)
                        /
                        len(X_label[:, feature_idx])
                        for category in
                              numpy.unique(X_label[:,feature_idx])
                    }
                    for feature_idx in range(X_label.shape[1])
                ]

                self._probs[index] = feature_probs

    def _gauss(self, class_index: int, x: numpy.ndarray):
        mean = self._mean[class_index]
        var = self._var[class_index]

        return numpy.exp(-((x - mean) ** 2) / (2 * var)) /
               numpy.sqrt(2 * numpy.pi * var)

    def _predict(self, x: numpy.ndarray):
        # In order to avoid overflow and underflow problems, log likelihood
        # is used

        if self.method == "gaussian":
            posteriors = [
                numpy.log(self._prior[index]) # Prior
                +
                numpy.sum(numpy.log(self._gauss(index, x))) # Posterior
                for index in range(len(self._classes))
            ]

        elif self.method == "categorical":
            def get_posterior(class_index: int, feature_index: int):
                try:
                    return self._probs[
                              class_index
                           ][feature_index][x[feature_index]]
                except KeyError as ex:
                    return 1    # log(1) = 0

            posteriors = [
                numpy.log(self._prior[index]) # Prior
                +
                numpy.sum(
                  [
                        numpy.log(get_posterior(index, feature_index))
                        for feature_index in range(len(x))
                  ]
                )    # Posterior
                for index in range(len(self._classes))
            ]

        return self._classes[numpy.argmax(posteriors)]

    def predict(self, X: numpy.ndarray):
        predictions = [self._predict(x) for x in X]
        return predictions
```

Appendix H

Logistic Regression's Training Loop on Breast Cancer Dataset:

```
from algorithms.logistic_regression import LogisticRegression

from src.utils.globals import Globals
from src.utils.logger import Logger

from algorithms.commons.evaluators import f1_macro, confusion_matrix,
plot_loss, plot_f1

import numpy

def breast_cancer(self):
        from sklearn.datasets import load_breast_cancer
        dataset = load_breast_cancer()

        features = dataset.data
        labels = dataset.target

        features_train, features_validation, features_test =
                                self.preprocessor.timeseries_split(features)
        labels_train, labels_validation, labels_test =
                                self.preprocessor.timeseries_split(labels)

        feature_batches = self.preprocessor.mini_batch(features_train)
        label_batches = self.preprocessor.mini_batch(labels_train)

        class_weights = None
        if self.config_service.logistic_regression_class_weights:
            from sklearn.utils.class_weight import compute_class_weight
            class_weights = compute_class_weight(
                                    class_weight="balanced",
                                    classes=numpy.unique(labels_train),
                                    y=labels_train
            )

        model = LogisticRegression(
    num_features=features.shape[1],
    regularization=self.config_service.logistic_regression_regularization,
    constant=self.config_service.logistic_regression_constant,
    stochastic=self.config_service.logistic_regression_stochastic,
    class_weights=class_weights
        )

        learning_rate =
                    self.config_service.logistic_regression_learning_rate
        training_loss_per_epoch = []
        validation_loss_per_epoch = []
        validation_f1_score_per_epoch = []
        for epoch in range(
                  1, self.config_service.logistic_regression_num_epochs+1):
            training_loss = 0
            for batch_index in range(len(feature_batches)):
                feature_batch = feature_batches[batch_index]
                label_batch = label_batches[batch_index]
```

```
                training_loss += model.fit(
                                feature_batch, label_batch, lr=learning_rate
                )

            training_loss /= batch_index+1

            validation_preds, validation_loss = model.predict(
                        features_validation,
                        self.config_service.logistic_regression_threshold,
                        labels_validation
            )
            validation_f1 = f1_macro(labels_validation,
                                                numpy.array(validation_preds))

            Logger.info(f"[Logistic Regression] Epoch: {epoch} | Training\
Loss: {training_loss} | Validation Loss: {validation_loss} | Validation F1\
Score: {validation_f1}"
            )
            training_loss_per_epoch.append(training_loss)
            validation_loss_per_epoch.append(validation_loss)
            validation_f1_score_per_epoch.append(validation_f1)

        with numpy.printoptions(threshold=numpy.inf):
            Logger.info(f"[Logistic Regression] Weights: \n{model.weights}"
            )

        test_preds = model.predict(
                        features_test,
                        self.config_service.logistic_regression_threshold
         )
        Logger.info(
                        f"[Logistic Regression] Test F1 Score: {f1_macro(
                        labels_test, numpy.array(test_preds)
                        )}"
        )
        Logger.info(f"[Logistic Regression] Test Confusion Matrix: \n
                  {confusion_matrix(
                  labels_test,
                  numpy.array(test_preds)
            )}"
        )

        plot_loss(
            training_loss=training_loss_per_epoch,
            validation_loss=validation_loss_per_epoch,
            num_epoch=self.config_service.logistic_regression_num_epochs,
            savefig_path=Globals.artifacts_path.joinpath("loss.png")
        )
        plot_f1(
            f1_scores=validation_f1_score_per_epoch,
            num_epoch=self.config_service.logistic_regression_num_epochs,
            savefig_path=Globals.artifacts_path.joinpath("f1.png")
        )

        self.save_configs()
```

Appendix I

K Nearest Neighbors Algorithm's Training on Breast Cancer Dataset:

```
from algorithms.k_nearest_neighbors import KNearestNeighbors

from src.utils.logger import Logger

from algorithms.commons.evaluators import f1_macro, confusion_matrix

import numpy

def breast_cancer(self):
    from sklearn.datasets import load_breast_cancer

    dataset = load_breast_cancer()

    features = dataset.data
    labels = dataset.target

    features_train, features_validation, features_test =
                                self.preprocessor.timeseries_split(features)
    labels_train, labels_validation, labels_test =
                                self.preprocessor.timeseries_split(labels)
    features_train = numpy.concatenate(
                                      (features_train, features_validation)
    )
    labels_train = numpy.concatenate((labels_train, labels_validation))

    model = KNearestNeighbors(k=self.config_service.k_nearest_neighnors_k)

    # Train model
    model.fit(features_train, labels_train)

    # Test Model
    similarity_measure =
            self.config_service.k_nearest_neighnors_similarity_measure
    predictions = model.predict(
                  features_test, similarity_measure=similarity_measure
    )

    test_f1 = f1_macro(labels_test, numpy.array(predictions))
    test_confusion_matrix = confusion_matrix(
                                       labels_test, numpy.array(predictions))
    Logger.info(f"[K Nearest Neighbor] Test F1 Score: {test_f1}")
    Logger.info(f"[K Nearest Neighbor] Test Confusion Matrix: \n
                                                {test_confusion_matrix}")

    self.save_configs()
```

Appendix J

Naïve Bayes Algorithm's Training on Breast Cancer Dataset:

```
from algorithms.k_nearest_neighbors import NaiveBayes

from src.utils.logger import Logger

from algorithms.commons.evaluators import f1_macro, confusion_matrix

import numpy

def breast_cancer(self):
    # Read data
    from sklearn.datasets import load_breast_cancer

    dataset = load_breast_cancer()

    features = dataset.data
    labels = dataset.target

    # Split data
    features_train, features_validation, features_test =
                                self.preprocessor.timeseries_split(features)
    labels_train, labels_validation, labels_test =
                                  self.preprocessor.timeseries_split(labels)
    features_train = numpy.concatenate(
                                        (features_train, features_validation))
    labels_train = numpy.concatenate((labels_train, labels_validation))

    # Initialize Model
    model = NaiveBayes(method=self.config_service.naive_bayes_method)

    # Train Model
    model.fit(features_train, labels_train)

    # Test & Evaluate Model
    predictions = model.predict(features_test)

    test_f1 = f1_macro(labels_test, numpy.array(predictions))
    test_confusion_matrix = confusion_matrix(
                                       labels_test, numpy.array(predictions))
    Logger.info(f"[Naive Bayes] Test F1 Score: {test_f1}")
    Logger.info(f"[Naive Bayes] Test Confusion Matrix:
                                                \n{test_confusion_matrix}")

    self.save_configs()
```