Ensembling Techniques on Deep Learning Models for Jane Street Market Prediction Analysis

Zeng Fung Liew Jake Cheng Hyeonji Choi

March 19, 2021

Abstract

Stock trading is an exciting challenge to take on due to its highly volatile nature that comes with potentially high returns. In this project, we approached this decision problem using various machine learning algorithms. Our results showed that traditional machine learning algorithms generally did not work well as compared to deep learning models, with our resulting average test accuracies being 54% and 59.9% respectively. Even then, the predictions made by deep learning models were still unsatisfactory. To further improve the prediction accuracies obtained by deep learning models, we transformed the input signals using Principal Component Analysis (PCA), a feature extraction technique. The outputs of the new best deep learning models based on the transformed data were then fed into a stacking ensemble to fit secondary classifiers. This led to a significant improvement in prediction accuracies, with our best test accuracy being 66.64%.

1 Introduction

The dataset for our project is the Jane Street market data from Kaggle. Using the market data from the major global stock exchange, our objective was to generate quantitative trading models to decide whether or not to trade a market stock in order to maximize returns. As financial markets are not perfectly efficient and fast-moving, we were confronted with highly volatile dataset which made the problem extremely challenging.

2 Methodology

2.1 Description of Data Set

This dataset contains an anonymized set of 130 time-series features representing normalized real stock market data, along with their respective (non-normalized) weights. In the `train.csv` dataset, we were also given an additional 5 response variables, each representing the returns over different time horizons. However, in order to make a trading decision that maximizes the return, some form a data transformation was required. Here we obtained the first component of the 5 response variables using Principal Component Analysis (PCA), and converted the result into binary values (ie. `1` if result is larger than zero, and `0` otherwise).

The `train.csv` file contained more than 2 million rows of data. After clearing out the rows containing `NaN` values or having zero weight, we were left with about 1.6 million rows of data. These were then randomly split into 80% train and 20% test data, where we assume the test data as unseen data. This was obtained by setting `random_state=1` throughout the entire project. All machine learning algorithms performed in this project were done solely on the train set.

2.2 Description of Algorithm

Due to the nature of this data set, finding a good classification algorithm that generalizes well was a difficult task. Traditional classifiers such as Support Vector Machines (SVM) and Classification Trees (CT) only managed to yield accuracy of at most 57%, which were far from satisfactory. Further details for these algorithms are discussed under "Additional Algorithms and Results" in Section 6.

To improve on this test accuracy, deep learning models were implemented. It turned out that for this data set, deep learning models tend to do better than the traditional machine learning algorithms with an average test accuracy of 59.9%.

After numerous testing, the best model was found to be a model where the top-57 components (accounting for 99% of variability) of the input time-series data from Principal Component Analysis (PCA) is passed through a deep learning model with 6 hidden layers. This model managed to give us up to 64.1% accuracy on the test data, which was a huge improvement. However, this model can be further improved by training multiple models of the same architecture before feeding their outputs into another classifier. This was possible due to the fact that neural networks are non-convex models, which means that randomly assigned weights prior to the start of the model training procedures lead to solutions of different local minima. By training the model five times and then feeding the outputs into a traditional machine learning classifier or a voting step, the resulting test accuracy was bumped up to 66.7%. Specific details of both of these models are further discussed in the Section 3.

3 Implementation Details

As mentioned previously, our best performing model was split into two steps: deep learning and stacked ensembling/voting. While the ensembling/voting procedure managed to give our model an improved prediction, such accuracy is not attainable without a good deep learning model.

3.1 Step 1: Deep Learning

The architecture of our deep learning model is shown in Figure 1. As mentioned in the previous section, PCA was first conducted on the time series data before being passed into the deep learning model. The reason for doing this instead of fitting in the entire time series is that the time series contains a lot of white noises, which could disturb the training of our model. In fact, taking the first 57 PCA components of the time series was sufficient in obtaining its most important features. However, note that PCA was only computed on the training data set and not the test data set. The dimensionally reduced test data was obtained via the projections of the first 57 principal components from the training data.

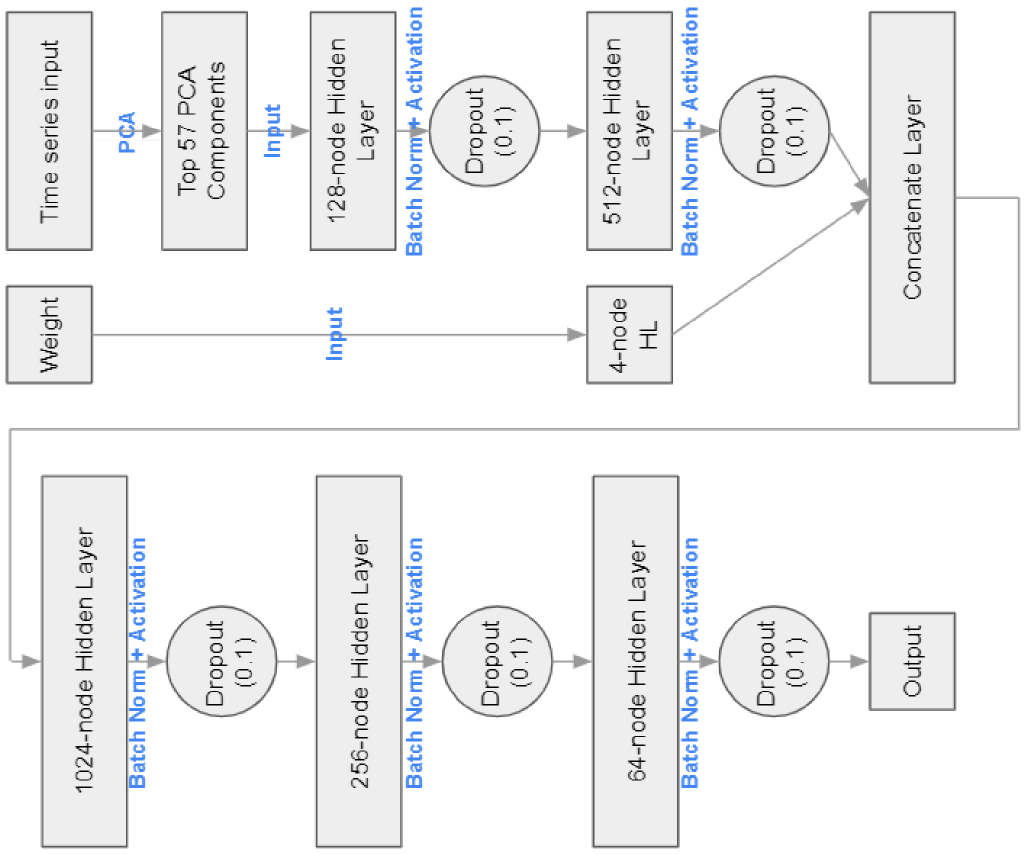


Figure 1: Architecture of the deep learning model

As for the number of nodes and hidden layers for this architecture, they were simply the result of trial-and-errors that turned out to work well. However, it is important to note that initial tests on this deep learning architecture found that the test accuracies were significantly lower than the training accuracies ($\sim 83\%$ train accuracy vs $\sim 60\%$ test accuracy). This was caused by the overfitting of our training dataset, and we overcame this by adding dropout onto each layer with a rate of 0.1. This led to 10% of randomly selected weights to be ignored during each EPOCH of the training phase. The idea behind this is that the classification of our data should not be heavily dependent on a few specific nodes, and by randomly ignoring weights in the training phase, we forced other nodes to carry more "responsibilities" in the classification step.

Additionally, batch normalization was added on to each hidden layer to accelerate the training procedure. To put it simply, this step normalizes our data at each hidden layer so that they follow the same distributions at all time and prevent the internal covariate shift phenomenon [8]. Other parameters that were set in building this model were relatively trivial, with all activation functions (except for the output layer being sigmoid) being Rectified Linear Units (ReLU). The optimizer used here was nesterov accelerated Stochastic Gradient Descent (SGD) with a learning rate of 0.5 and momentum of 0.99. To allow for more optimized learning near the local minima, the learning rate decay was set to be 0.0005. During the training phase, 20% of the training data was randomly selected and used as a validation set.

3.2 Step 2: Stacking Ensemble

As mentioned previously, due to the non-convex nature of deep learning models, each training of the exact same model produces different results. Therefore, setting up a 2nd level classifier/voting system based on the outputs of the previous models can be thought of as a type of ensembling method: each model's prediction is a "vote" and the weight of each "vote" can be different in achieving the final prediction. A basic flow chart of this technique is shown in Figure 2. It is important to note that in order to maximize the effect of this ensemble classifiers, the probability outputs of the deep learning models should be used instead of the predictions [10]. In this project, the 2nd level classifiers used were the majority and average votes, Gaussian Naive-Bayes, Logistic Regression, Linear Discriminant Analysis (LDA), Quadratic Discriminant Analysis (QDA), K-Nearest Neightbors (KNN), and Classification Trees (Decision Tree, Random Forest, AdaBoost, Gradient Boosting Classifier).

Among the 2nd level classifiers above, the majority and average voting classifiers are the most straight forward, and requires no training at all. In the former case, each model makes a prediction on the class of a given data. The predictions are then gathered and the class with the most votes becomes the final prediction. As for the latter case, each model outputs a probability of classifying the trading action to be `True`. The probabilities are averaged and the final prediction is set to `True` if the average probability is larger than 0.5, and `False` otherwise.

The other classifiers mentioned above are traditional machine learning classifiers. These classifiers take the outputs of the deep learning models as their inputs and undergo a secondary model fitting procedure before making the final predictions.

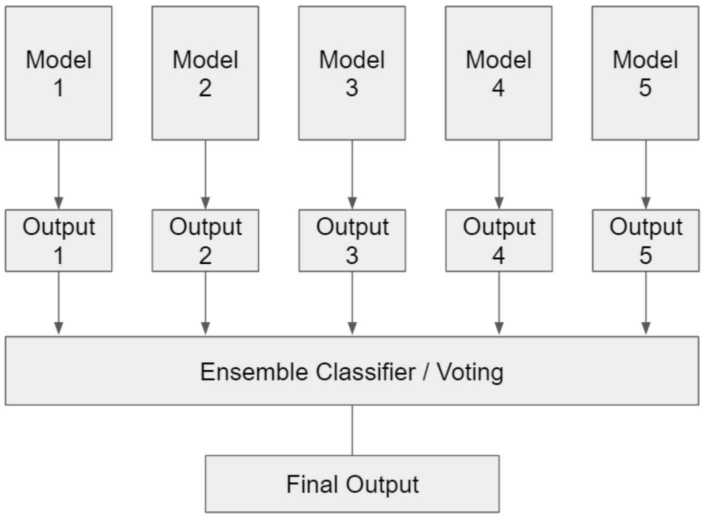


Figure 2: Flow chart of ensemble classifier/voting system based on the outputs of previous deep learning models.

However, using a machine learning classifier means that hyperparameters such as L2-regularizing constants or learning rates needed to be tuned to obtain a best generalizing classifier with the lowest possible prediction error. This was done using a grid search on the hyperparameters in question. To reduce the variance in our error rates, 5-fold Cross Validation was applied at each iteration of the grid search. The results of the Grid Search Cross Validation and the comparison between different ensemble classifiers are discussed in the Section 4.

4 Results and Interpretation

4.1 Deep Learning

The results for the model described in Section 3.1 is shown in Table 1. Recall that the same model is trained 5 times for the purpose of stacking ensemble as discussed in Section 3.2.

Model	Train(%)	Test(%)
Model 1	73.7	63.9
Model 2	74.4	63.8
Model 3	71.8	63.2
Model 4	74.5	63.8
Model 5	74.7	64.1

Table 1: Train and test set accuracies for 5 similarly trained deep learning models

The results show high similarities in prediction accuracies, except for the case of Model 3. Compared to the results from traditional classifiers, this above model performed significantly better. This came at the expense of model fitting time, as it took weeks to find a model architecture that produced desirable results with considerably less overfitting.

4.2 Stacking Ensemble

The results for the stacking ensemble of models described in Section 3.2 are shown in Table 2, and its comparison with the deep learning models are shown in Figure 3.

Ensemble	Best Scikit-Learn Parameters	Train(%)	Test(%)
Majority Vote	-	77.89	66.15
Average Vote	-	79.13	66.61
Gaussian Naive-Bayes	-	79.42	**66.64**
Logistic Regression	`C:10, l1_ratio:0.5`	79.55	66.57
LDA	-	79.31	66.43
QDA	-	78.83	66.27
KNN	`n_neighbors:500, weights:"distance"`	99.99	66.60
Decision Tree	`criterion:"gini", min_samples_split:200, splitter:"random"`	80.05	66.23
Random Forest	`min_samples_split:200, n_estimators:100`	82.32	66.47
Gradient Boosting	`learning_rate:1, min_samples_split:2, n_estimator:50`	79.37	66.51
AdaBoost	`learning_rate:1, n_estimator:500`	79.47	66.52
SVM	`kernel:"rbf", max_iter:1e4, C:0.1`	52.86	51.68

Table 2: Accuracies of stacking ensembles, along with their best parameters

Overall, we observed that the stacking ensembles manage to boost our test accuracies by about 2-3%, which was highly significant considering how low our previous prediction accuracies were. Interestingly, in this case, simple classification methods seemed to work best for stacking ensemble. In Table 2, we see that Naive-Bayes classifier and average voting ensemble produced the best results with some of the shortest training times (only majority vote is arguably faster). On the other hand, we note that having SVM as our 2nd level classifier seemed to produce worse results, though this issue might have arose due to a small number of computed iterations.

Other machine learning methods such as Logistic Regression, KNN, and Classification Trees showed promising results, as their test accuracies were really close to that of the Naive-Bayes. However, due to the time constraint for this project, it was not possible to obtain the best sets of tuning parameters as each model fitting iteration took about 4-5 hours (SVM took days per iteration!) due to the large data size. Longer training and grid searching over larger amount of parameters for these classifiers could possibly improve the test accuracies even more.

5 Conclusions

Throughout this project, we mainly observed the strength of stacking ensembles in improving a classifier. With a set of deep learning models that achieved at most 64% test accuracy, we were able to improve the classifiers by up to to 3%. An interesting finding here is that simple models like the voting methods and Naive-Bayes turned out to be more impactful that several other classifiers. However, we do need to note that the difference is not very significant, and that implementations of different primary models with different secondary classifiers could yield different results. The important takeaway here is that average voting on stacking ensembles is a good solution if time is an issue.

One mistake made in this project was the use SVM as our primary classifier to gauge the predictive abilities of Machine Learning models. The training of an SVM model has a time complexity of $O(nd^2)$ [3] as compared to lightweight models such as Logistic Regression ($O(d)$) [4] and was not an efficient model to use. Additionally, the feature engineering process is one that cannot be neglected. As we will discuss in Section 6.3, the success of the model in Section 3.1 was largely due to the feature extraction step which is PCA.

In addition, we also discovered an improved implementation of Adaboost. Instead of using the default Adaboost classification algorithm (using decision trees of depth 1 as the base estimator), we added grid search cross validation in both steps and compare with our results from regular decision tree classifier. This resulted in a boost in test accuracy from 51% to 57%. However, a decision tree was not the only option in this boosting process. In fact, the improved Adaboost can be used on any weak learner algorithms, such as Logistic Regression, Naive-Bayes, and Random Forest [1].

6 Additional Algorithms and Results

As mentioned in Section 2.2, traditional classifiers did not generate ideal test accuracies for further implementations. However, it's crucial to discover the reason of the unsatisfactory performance. In the following subsection, the methodology and test accuracies

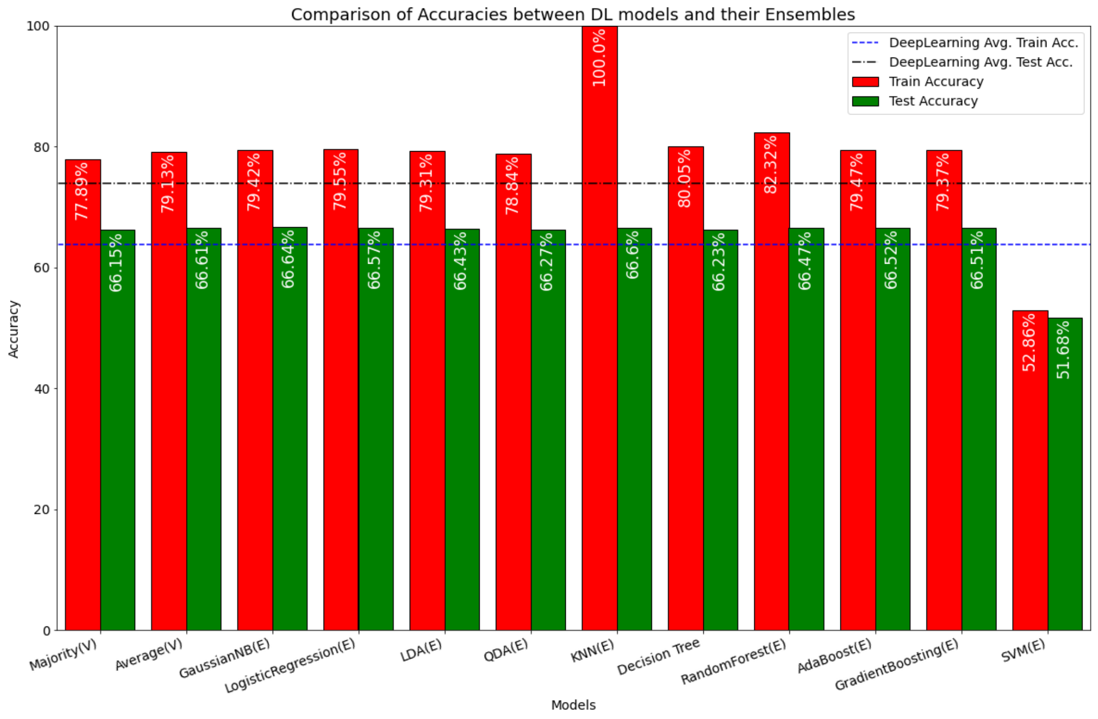


Figure 3: Comparison of stacking ensembles with the original deep learning models

of six different classification models are going to be discussed. The results for all the models discussed in Section 6 are shown in Table 3.

6.1 Traditional Classifiers

1. Logistic Regression: Logistic Regression is known as a go-to classification model for binary classification problems. However, without tuning any parameters, the test accuracy was around 53% along with a warning message about reaching maximum iteration. The low accuracy performance suggested that the hyper-parameters should be tuned through cross validation before the model fitting process. Usually, Logistic Regression converges very quickly even with the low default number of iterations, so tuning the iteration numbers might help to resolve the warning message. We set the penalty term to Elastic Net, solver to Saga and iteration number to 1000, along with the regularization strength parameter `C` and L1 ratio in the grid search process.

2. Support Vector Machine (SVM): SVM is a powerful tool for classification problems due to its low tendency of overfitting, robust implementations with kernel tricks, and its definition as a convex problem [2]. The objective of this methodology is to find a hyper plane in an n-dimensional space that separates the data points to their potential classes. In this sense, the kernel functions calculate the distance between data points by transforming the data points to a higher-dimensional mapping, which reduces the computational effort and time [12]. However, the SVM algorithm, with a time complexity of $O(nd^2)$, comes at a cost of high computation time in large datasets, and the training time in this case was more than a week! With the implementation of the RBF kernel, the `gamma` and regularization parameter `C` were tuned in the grid search process.

3. Decision Tree Classifier : Tree-based methods are simple and easy to interpretation. However, they are not as competitive as the deep learning models in Section 3.1. In classification trees, we predict that each observation belongs to the most commonly occurring class of training observations in the region to which it belongs. Unfortunately, trees generally do not have the same level of predictive accuracy as some of the other classification approaches. Therefore, parameters such as criterion, splitting strategy and the minimum number of sample split were tuned using grid search cross validation to improve our test accuracy.

4. Random Forest : As mentioned in the lecture notes, a random forest is essentially a collection of decision trees. To avoid low accuracy performances, grid search 5-fold cross validation was implemented to tune parameters such as splitting criterion and minimum sample split size.

5. Adaboost : Boosting is a general ensemble method that creates a strong classifier from a number of weak classifiers. Adaboost, short for adaptive boosting, is known as a successful boosting algorithm developed for binary classification. Adaboost is best known for boosting the performance of decision trees. Nevertheless, Adaboost can be used to boost the performance of any machine learning algorithms [5]. It is best used with weak learners, which are models that achieve accuracies just above random chance on a classification problem.

6. Regular Adaboost: The default base estimator for the regular Adaboost implementation is single-level decision trees (aka. stumps) [6]. Here, grid search cross validation was only implemented on the number of estimators.

7. Adaboost-Decision Tree Hybrid Classifier: Instead of using the default settings for Adaboost algorithm, we combined it with decision trees of multiple levels (such as the ones in (3)) to boost its performance. Here, grid search cross validation was implemented in search of the best criterion and splitting strategy of decision trees, along with the number of estimators used for Adaboost.

6.2 Other Deep Learning Models

While the model described in Section 3.1 was the most successful, there were other notable deep learning models that did really well. These models are briefly described below:

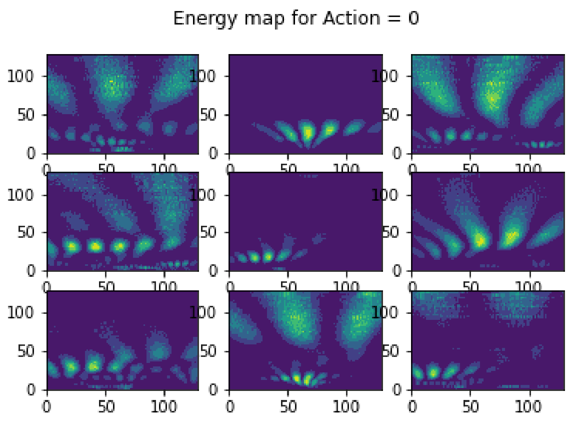
1. Simple feed-forward model 1 (FF1): All 130 time series data + weight variable were passed through a total of 6 hidden layers with batch normalization and dropout. The Adam optimizer with learning rate of 0.01 was used for gradient descent.
2. Deep feed-forward model 1 (FF2): Similar to FF1, but this time the data was passed through a total of 12 hidden layers with batch normalization and dropout.
3. Concatenation of 2 simple feed-forward model 1 (CFF1): The 130 time series data was passed through 4 hidden layers, while the weight variable was passed through 1 hidden layer. Both outputs are concatenated and passed through another 2 hidden layers. The Adam optimizer with learning rate of 0.01 was used for gradient descent.
4. Concatenation of 2 feed-forward model 2 (CFF2): Similar to CFF1, but this time the 130 time series data was passed through 8 hidden layers, and the outputs were concatenated and passed through another 4 hidden layers. The Adam optimizer with learning rate of 0.01 was used for gradient descent.
5. Concatenation of Convolutional-Feedforward model 1 (CCFF1): The 130 time series data was passed through 4 convolutional layers with max pooling, while the weight variable was passed through 1 hidden layer. Both outputs were concatenated and passed through for another 9 hidden layers. The Adam optimizer with learning rate of 0.01 was used for gradient descent.
6. Concatenation of Convolutional-Feedforward model 2 (CCFF2): Similar to CCFF1, but this time there were 13 convolutional layers used for the time series data, and only 2 hidden layers after concatenation. The nesterov boosted Stochastic Gradient Descent optimizer with learning rate of 0.1, momentum of 0.9 was used for gradient descent. This design was inspired by the VGG-16 architecture [13].
7. Concatenation of Convolutional-Feedforward model 3 (CCFF3): Similar to CCFF1, but this time there were 2 hidden layers used after concatenation. The Stochastic Gradient Descent optimizer with learning rate of 0.05 and momentum of 0.9 was used for gradient descent. This design was inspired by the AlexNet architecture [11].
8. PCA + Concatenation of 2 feed-forward model 1 (PCFF1): The top-57 components of the PCA'ed time series was passed through 3 hidden layers with batch normalization and dropout, while the weight variable was passed through 1 hidden layer. Both outputs were concatenated and passed through another 3 hidden layers. The nesterov boosted Stochastic Gradient Descent optimizer with learning rate of 0.5 and momentum of 0.9 was used for gradient descent.
9. PCA + Concatenation of 2 feed-forward model 2 (PCFF2): Similar to PCFF1, but this time the PCA'ed time series was passed through 2 hidden layers, and the concatenated results were passed through another 2 hidden layers. The Stochastic Gradient Descent with learning rate of 1 and momentum of 0.9 was used for gradient descent.

Compared to the selected model in Section 3.1, these models gave us different characteristics of results. In general, the resulting test accuracies were lower than the best model. However, there were some promising models that were severely overfitted, yet refitting those models with different regularization parameters were not possible due to extremely long training time. An example of this is the CCFF2 model, which took over 5 days to train. The complete results of these models are shown in Table 3 and discussed in a later section.

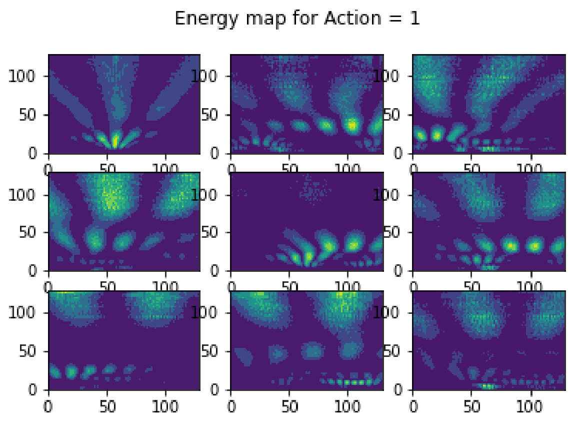
6.3 Miscellaneous

An important process when working in this project is feature engineering. This is a step where important features were extracted to help distinguish the two classes. Some of the feature engineering methods that we tried in this project were inspired by Taspinar's work [14], which include signal denoising by thresholding and feature extractions using discrete wavelet transforms, and the construction of scaleograms using continuous wavelet transforms.

For the denoising of signals, each signal was decomposed into scaling and detail coefficients by passing through high and low-pass filters. These decomposed signals were then thresholded with a small value which was determined using the relative error curve [9]. This resulted in signals void of their white noise. However, the process of denoising turned out to be counterintuitive, as it led to a reduction in accuracy in our models. Similarly, extracting features such as entropy, summary statistics and zero crossings rate, and also the construction of scaleograms failed to produce any meaningful results. As observed in Figure 4, the scaleograms did not show much distinctions, which explains why obtaining a desirable result was such a challenge.



((a)) Scaleograms for class action=0



((b)) Scaleograms for class action=1

Figure 4: Scaleograms of randomly selected signals

With all the unsuccessful feature extraction techniques so far, we turned our attention to a simple dimension reduction strategy, which was the Principal Component Analysis (PCA). Fortunately, this turned out well as we hoped. Due to the lack of time, other potentially useful feature extraction methods for signal processing such as the independent component analysis (ICA), kernel PCA, robust PCA, local discriminant basis (LDB) and autoencoders were not experimented.

6.4 Overall Results

Classifier Model	Best Scikit-Learn Parameters	Train(%)	Test(%)
Logistic Regression	`C:1, l1_ratio:0.5`	53.01	52.99
SVM	`kernel:"rbf", C:0.1`	57.01	52.98
Decision Tree	`criterion:"gini", min_samples_split:2, splitter:"best"`	100	51.19
Random Forest	`criterion:"entropy", min_samples_split:2`	100	60.49
AdaBoost	`n_estimator:500`	53.73	53.42
Adaboost-Decision Tree Hybrid	`criterion:"gini", splitter:"best", n_estimators:500`	100	57.17
FF1	-	0.59	0.57
FF2	-	0.64	0.6
CFF1	-	0.61	0.58
CFF2	-	0.6	0.58
CCFF1	-	0.74	0.54
CCFF2	-	0.9	0.577
CCFF3	-	0.9	0.55
PCFF1	-	0.73	0.61
PCFF2	-	0.58	0.596

Table 3: Accuracies of all other experimented classification models, along with their best parameters

For our traditional classifiers, random forest achieved the best test accuracy score as it added randomness to the model while growing the decision trees. Instead of searching for the most important feature while splitting a node, it searched for the best feature among a random subset of features. Since the stock market data was very unpredictable, it generated a better fit. However, logistic regression, SVM and decision trees did not achieve satisfactory performances. For decision trees, overfitting was a huge issue as a train accuracy of 100% was constantly obtained, yet the test accuracy had a performance similar to random chances. This is due to the deterministic and extremely greedy nature of decision trees as they seldom provide the predictive accuracies comparable to the best that can be achieved with the test data [7]. However, boosting decision trees can dramatically improve their accuracies, which led to the exploration of Adaboost-decision tree hybrid method in part 5 in Section 6.1. Additionally, the low test accuracy for logistic regression suggested that the classes were not linearly separable.

As for the deep learning models, it was interesting to see that the convolutional neural network models did not perform as well as hoped. These models were expected to produce superior results due to the ability of convolutional filters to pick up patterns in signals. Once again, this could be due to the volatility of stock markets, which caused signals to be extremely noisy. However, the fact that denoising the signals using wavelet transform gave us worse results seem to have contradicted that inference. One thing that can be concluded here is that the features of the signals from both classes are extremely similar, and one ought to have better feature engineering knowledge to tackle this problem and improve on the current results.

To prove the importance of feature engineering in this problem, we look no further than the current feature extraction technique that we used in this project. The models PCFF1 and PCFF2, along with the best model used in Section 3.1, all went through the PCA feature extraction process. This did not only reduced the dimensions of the problem, but also sped up the training process. In fact, a better denoising could also have been applied onto SVM to improve its predictive ability.

Resources

The detailed codes for reproducing the results can be obtained on Zeng Fung's Github repository here.

References

[1] Matt Krause (https://stats.stackexchange.com/users/7250/matt-krause). *What is meant by 'weak learner'?* Cross Validation. eprint: `https://stats.stackexchange.com/q/82063`. URL: `https://stats.stackexchange.com/q/82063`.

[2] Dikran Marsupial (https://stats.stackexchange.com/users/887/dikran-marsupial). *Advantages and disadvantages of SVM.* eprint: `https://stats.stackexchange.com/q/24440`. URL: `https://stats.stackexchange.com/q/24440`.

[3] Alekhyo Banerjee. *Computational Complexity of SVM.* Aug. 2020. URL: `https://alekhyo.medium.com/computational-complexity-of-svm-4d3cacf2f952`.

[4] Writuparna Banerjee. *Train/Test Complexity and Space Complexity of Logistic Regression.* Aug. 2020. URL: `https://levelup.gitconnected.com/train-test-complexity-and-space-complexity-of-logistic-regression-2cb3de762054`.

[5] Jason Brownlee. *Boosting and AdaBoost for Machine Learning.* Aug. 2020. URL: `https://machinelearningmastery.com/boosting-and-adaboost-for-machine-learning/`.

[6] Yoav Freund and Robert E Schapire. "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting". In: *Journal of Computer and System Sciences* 55.1 (1997), pp. 119–139. ISSN: 0022-0000. DOI: `https://doi.org/10.1006/jcss.1997.1504`. URL: `https://www.sciencedirect.com/science/article/pii/S002200009791504X`.

[7] Trevor Hastie, Jerome Friedman, and Robert Tisbshirani. *The Elements of statistical learning: data mining, inference, and prediction.* Springer, 2017.

[8] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167 (2015). arXiv: `1502.03167`. URL: `http://arxiv.org/abs/1502.03167`.

[9] Jeff Irion and Naoki Saito. "Efficient Approximation and Denoising of Graph Signals Using the Multiscale Basis Dictionaries". In: *IEEE Transactions On Signal And Information Processing Over Networks* (). URL: `https://escholarship.org/content/qt0bv9t4c8/qt0bv9t4c8_noSplash_66d3d84d7c4f3146a80f5611e0214b1b.pdf`.

[10] K.M.Ting and I.H.Witten. "Issues in Stacked Generalization". In: *Journal of Artificial Intelligence Research (JAIR)* 10 (1999). DOI: `https://doi.org/10.1613/jair.594`. URL: `https://www.jair.org/index.php/jair/article/view/10228`.

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems.* Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: `https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf`.

[12] Hucker Marius. "Multiclass Classification with Support Vector Machines (SVM), Dual Problem and Kernel Functions". In: *Towards Data Science* (June 2020). URL: `https://towardsdatascience.com/multiclass-classification-with-support-vector-machines-svm-kernel-trick-kernel-functions-f9d5377d6f02`.

[13] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition.* 2015. arXiv: `1409.1556 [cs.CV]`.

[14] Ahmet Taspinar. *A guide for using the Wavelet Transform in Machine Learning.* Dec. 2018. URL: `https://ataspinar.com/2018/12/21/a-guide-for-using-the-wavelet-transform-in-machine-learning/`.