Department of Electrical Engineering & Computer Science

Final Project - Group 4

Course: EECS 4404/5327

Instructor: Dr. Ruth Urner

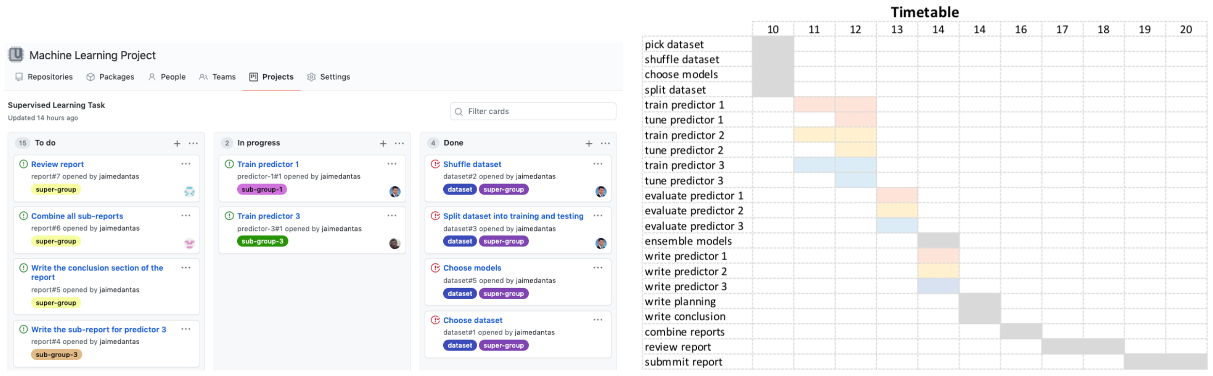
Due Date: Sunday, December 20, 2020

Name	**Student #**
Baoqi Yu	0
David Mounes	0
Jaime Dantas	0
Matthew Tolentino	0
Minghong Xu	0
Nizwa Javed	0
Thomas Lykouras	0
Yiming Shao	0
Umer Qudrat	0

December 17, 2020

Planning and execution

In order to better organize the workflow of the project, we created a project board on GitHub to keep track of all tasks involved. We split the project into 21 distinct tasks, and some of the tasks can be seen in the left picture of figure 1.



Machine Learning Project

Repositories Packages People Teams Projects Settings

Supervised Learning Task
Updated 14 hours ago

Filter cards

15 To do

- Review report — report#7 opened by jaimedantas — super-group
- Combine all sub-reports — report#6 opened by jaimedantas — super-group
- Write the conclusion section of the report — report#5 opened by jaimedantas — super-group
- Write the sub-report for predictor 3 — report#4 opened by jaimedantas — sub-group-3

2 In progress

- Train predictor 1 — predictor-1#1 opened by jaimedantas — sub-group-1
- Train predictor 3 — predictor-3#1 opened by jaimedantas — sub-group-3

4 Done

- Shuffle dataset — dataset#2 opened by jaimedantas — dataset super-group
- Split dataset into training and testing — dataset#3 opened by jaimedantas — dataset super-group
- Choose models — dataset#5 opened by jaimedantas — dataset super-group
- Choose dataset — dataset#1 opened by jaimedantas — dataset super-group

Timetable

	10	11	12	13	14	14	16	17	18	19	20
pick dataset	▨										
shuffle dataset	▨										
choose models	▨										
split dataset	▨										
train predictor 1		🟥	🟥								
tune predictor 1			🟥								
train predictor 2		🟨	🟨								
tune predictor 2			🟨								
train predictor 3		🟦	🟦								
tune predictor 3			🟦								
evaluate predictor 1				🟥							
evaluate predictor 2				🟨							
evaluate predictor 3				🟦							
ensemble models					▨						
write predictor 1					🟥						
write predictor 2					🟨						
write predictor 3					🟦						
write planning						▨					
write conclusion						▨					
combine reports							▨				
review report								▨	▨		
submmit report										▨	▨

Figure 1: Project's task board on GitHub on the left and project's timetable on the right. Publicly available in `https://github.com/orgs/EECS-4404/projects/1`.
A repository for each predictor was created on GitHub, and a timetable (left picture of figure 1) was followed. We split the super-group into 3 sub-groups as shown below.

Sub-group 1	**Sub-group 2**	**Sub-group 3**
Jamie Dantas	Minghong Xu	Nizwa Javed
Matthew Tolentino	Baoqi Yu	Thomas Lykouras
David Mounes	Yiming Shao	Umer Qudrat

Each sub-group trained a different predictor. Additionally, the remaining tasks were equally distributed among everyone. Several Zoom calls were made to discuss the project, and a Discord channel was created for discussion.

Dataset

The dataset chosen was the Electrical Grid Stability Simulated Dataset from the UCI Repository [1]. It is a binary classification problem with 13 real inputs that represents simulations of stability of a given electrical grid, and it has 10,000 data points.

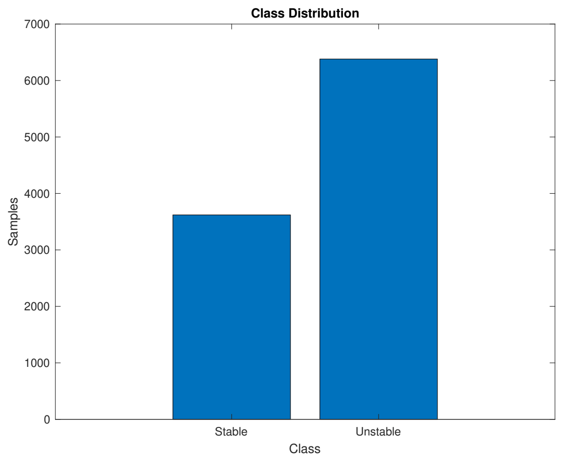


Figure 2: Class distribution of dataset. Publicly available in `https://github.com/EECS-4404/dataset`.

We performed a data pre-processing on the dataset to replace the class label "stable" and "unstable" to 1 and -1, respectively. The last input "stab" was removed since it contains the same information as the class label (stable if negative, unstable otherwise). Since this dataset is imbalanced, as shown in the figure 2, we used the F1-score for measuring the performance of the models. After pre-processing the dataset, we split it into a training dataset, with 80% of the data points, and a testing dataset, with 20% of the data points. The dataset was also shuffled. All the three predictors used the same data for training and testing.

Predictor #1: feed-forward neural network

The first predictor we trained was the feed-forward neural network. We used MATLAB for creating and tuning a pattern recognition neural network (*patternnet*). First, we transformed the last column of the dataset, i.e., the class label +1/-1, into two new columns filled with 1 in the index that represents the class and 0 otherwise [stable unstable]. Then, we tried a different set of combinations of hidden layers and neurons always aiming to simplicity over complexity. This process resulted in the creation of a neural network with two hidden layers with 36 and 24 neurons, respectively, as shown in the figure 3.

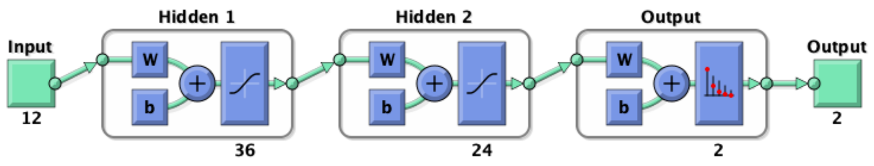


Figure 3: Feed-forward neural network created.

Note that the neural network created has 12 inputs and it outputs a 10000×2 vector with the class prediction. The final step of the tuning process was selecting the activation and performance function. For the transfer function, the best results were encountered with the symmetric sigmoid transfer function [3] in combination with the cross-entropy loss (logistic loss). The algorithm chosen for training was the scaled conjugate gradient backpropagation [4], and the neural network converged with approximately 100 epochs.

The regularization parameter λ of the loss function was set to 0.01 to present overfitting. Finally, we reserved 20% of the training dataset for validation. The figure 4 shows the performance and the confusion matrix for the training dataset.

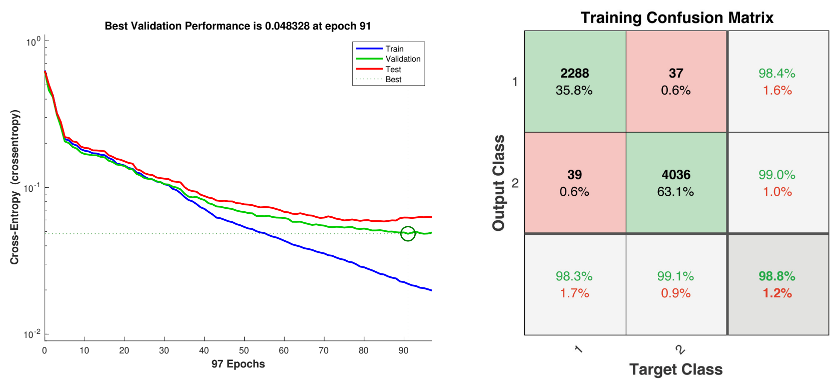


Figure 4: Performance of the neural network on the left, and the confusion matrix on the training dataset only on the right (1 stands for stable and 2 for unstable).

The F1-score of the training dataset is presented below.

$$F1_T = \frac{2 \cdot TP}{2 \cdot TP + FP + FN} = \frac{2 \cdot 2288}{2 \cdot 2288 + 39 + 37} = \mathbf{98.4\%}$$

When it comes to the F1-score on the validation dataset, we got $F1_V = 94.2\%$. Overall, this feed-forward neural network demonstrated a promising performance over this dataset.

Predictor #2: k-nearest neighbour predictor

The second predictor we trained was the k-nearest neighbour predictor. The process was completed in three steps: data preparation, parameter tuning and predictor evaluation.

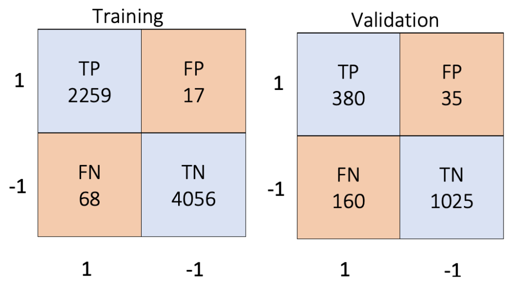
First of all, we trained our first demo predictor using the training dataset based on the KNN Scikit Tool (with default parameters) in Python [2]. After testing the predictor on the training data, we saw that the accuracy at this stage was 86.2%, which was not acceptable. So, we then tried a different range of k, from 1 to 50, and analyzed the results. The figure shows the results for the training dataset and the testing dataset using cross-validation. By observing the original data, we found that the range of each column in the input data are distinct, i.e., the range of tau1 (column 1) is around 0 to 10, and the range of g2 (column 10) is around 0 to 1. In this case, g2 may accidentally become more uninformative than tau1. This is because the calculation of distance in KNN is based on the input features, so a larger range feature has a larger impact on the distance since the Euclidian distance equation is given by $D(x,y) = \sqrt{\sum_n (x_i - y_i)^2}$. For this reason, we normalized all data by scaling them to the range of 0 to 1.

Then we split the training dataset into training (80%) and validation (20%), and obtained an accuracy of 86.7% on the validation dataset without any tuning. Next, we performed the parameter tuning step, which is one of the most important processes that dominate the behaviour of a predictor, and ultimately resulting in the best optimal combination of parameters possible. Although the Python sklearn's documentation states there are eight parameters, we only put three of them into our tuning process:

- *n_neighbours*: number of neighbours used to calculate k-neighbours. We tested from 2 to 20.
- weights: there are two choices: uniform or distance. Uniform means all points in each neighbourhood are weighted equally whereas distance means the closer point is weighted more. We tested both cases.
- *p*: Calculation procedure for the distance that the predictor will use: 1 for Manhattan distance, 2 for Euclidean distance and arbitrary number for Minkowski distance. We tried them all.

We used 30 as the default size for the *leaf_size* variable (related to the construction time of the predictor), and the default algorithm for the distance metric used for the tree.

Among all possible combinations, the one that had the best accuracy on training data was using *n_neighbours = 16, weights = distance*, and *p = 1.* The accuracy for this configuration was 87.9% for the validation dataset and 98.7% for the training dataset, which shows we did not overfit the data. We performed 10-fold cross-validation on the training dataset to avoid overfitting as much as possible, and the training and validation confusion matrices are presented in the figure 5.



Training

1	TP 2259	FP 17
-1	FN 68	TN 4056
	1	-1

Validation

1	TP 380	FP 35
-1	FN 160	TN 1025
	1	-1

Figure 5: Confusion matrix of the training dataset on the left, and for the testing dataset on the right.

We used the confusion matrix shown on figure 5 to calculate the F1-score for the training dataset.

$$F1_T = \frac{2 \cdot TP}{2 \cdot TP + FP + FN} = \frac{2 \cdot 2259}{2 \cdot 2259 + 17 + 68} = \mathbf{98.2\%}$$

When it comes to the F1-score on the validation dataset, we got $F1_V = 79.6\%$. In conclusion, the k-nearest neighbour predictor demonstrated satisfactory results in the training dataset.

Predictor #3: kernelized support vector machine

The third predictor we trained was the kernelized support vector machine (KSVM). To train and optimize the hyperparameters for the KSVM, we chose to use the classification learner toolbox in MATLAB [4]. Before training the predictor itself, we decided to use 5-fold cross-validation as a measure to prevent overfitting and to ensure we achieve a reliable predictor. This trade-off came at the expense of extra computing time when compared to using a holdout dataset.

After directly importing our training dataset into MATLAB, we had several options to go about training the model. The first one was the type of kernel to be used; we could either use the gaussian or polynomial kernels of degree N. Then, there were the hyperparameters that we could change which consisted of the box constraint for all kernels, the degree of polynomial for polynomial kernels, the kernel scale for gaussian kernels, and finally whether to normalize our input data. From there, we trained and cross-validated several models using linear, quadratic, cubic, and fine/medium/coarse gaussian kernels. After analyzing the results, we noticed that a cubic kernel outperformed the other kernels. To validate our findings, we then ran a Bayesian optimizer for 10 iterations to seek out the best hyperparameters that were previously mentioned.

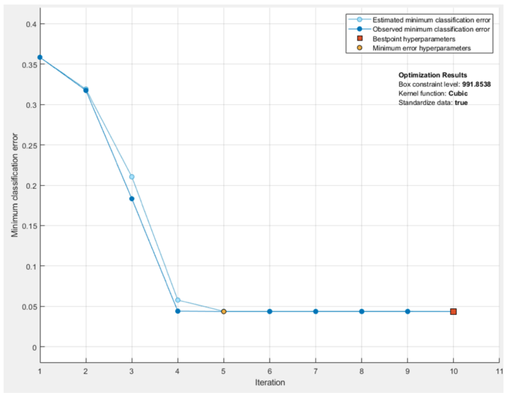


Figure 6: Hyperparameter optimization iterations process to achieve the lowest cross-validation loss.

This combination of the cubic kernel with a box constraint of 991.85, and normalized input data led to the best performance with the accuracy of 98.9%. Since this result was satisfactory, we then ran our predictor on the training and testing datasets to calculate the confusion matrices as shown in the figure 7.

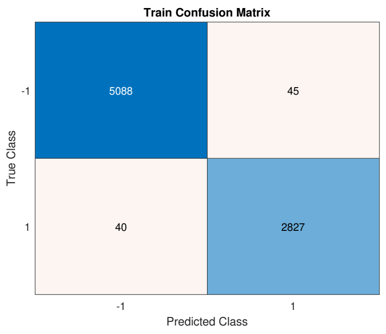


Figure 7: Confusion matrix of the training dataset.

From the confusion matrix in the figure 7, we can calculate the F1-score on the training dataset as follows:

$$F1_T = \frac{2 \cdot TP}{2 \cdot TP + FP + FN} = \frac{2 \cdot 2827}{2 \cdot 2827 + 45 + 40} = \mathbf{98.5\%}$$

From our results of a high cross-validation accuracy of 98.9% and an F1-score of 98.5% on our training data, it seems that our KSVM predictor achieves promising performance with little signs of overfitting.

Ensemble Learning

In this section, we have evaluated the three predictors on the test set. We also tried weighted majority voting for ensemble learning. For this purpose, we have used the prediction vector generated on the test set by each predictor. We have used Python and scikit learn for this task [2].

Standalone model accuracy and F1-score:
For each predictor, the accuracy and F1-score for the test set can be seen in the table below:

Predictor	**Accuracy**	**F-1 Score**
Feed Forward Neural Network	0.9545	0.939
KNN	0.861	0.7903
Kernelized SVM	**0.963**	**0.9506**

We can see that kernelized SVM outperforms the rest of the predictors on the test set. The second best performance is shown by the neural network. KNN seems to have the least accuracy and F1 score on the test set.

Weighted majority voting:
We combined the predictors using weighted majority hard voting. Each predictor was assigned a weight and all these predictions were combined by calculating dot product between assigned weight vector and prediction vector. We also created a grid search function to find a combination of weights that outputs the highest accuracy and F1-score. The results of some of the weights can be seen below:

Weights [Predictor 1, Predictor 2, Predictor 3]	**Accuracy**	**F-1 Score**
[0.333,0.333,0.333]	0.9595	0.9449
[0.25, 0.25, 0.5],[0.3,0.2,0.5]	**0.965**	**0.9538**
[0.5,0,0.5]	0.963	0.952

The first row shows results of assigning equal weights to all predictors. We see that the F1-score and accuracy is still less than standalone kernelized SVM. Second and third row show us weights generated by grid search which yield the best result. This is slightly higher than our best performing model (Kernelized SVM). Hence, we can see that ensemble learning can help in improving performance.

Results

The F1-score was used to evaluate the weighted average of precision and recall for the **feed-forward neural network** over this dataset due to the uneven class distribution. Applying the formula for the F1-score for the training, validation, testing, and all confusion matrices we get 98.4%, 94.2%, **93.9%**, and 96.8% respectively. These values are reasonable as the F1-score of the validation confusion matrix is less than the F1-score of the training-confusion matrix after adjusting the weights of the neural network to prevent overfitting. The results of the F1-score for the test confusion matrix is relatively close to the validation confusion matrix which shows that the validation predictor was successful.

When evaluating the performance of the **kernelized support vector machine**, we had an F1-Score of 98.5% and a 98.9% accuracy for our training dataset. When using the trained model to predict the labels for the testing dataset we had set aside, we calculated a new F1-score of **95.1%**. This result shows that the KSVM model performs well in predicting unseen data as careful thought went into ensuring that it did not overfit the training dataset.

On the other hand, even though the **k-nearest neighbour** predictor has shown an excellent accuracy on the training data, the results obtained on validation and testing were not as good as we expected. This is because the F1-score for the validation and testing dataset was only 79.6% and **79%**, respectively. When we compare these results to the training dataset (98.5%), we can notice that the true performance was far lower than the one obtained when training the predictor. One of the reasons for such a low performance can be overfitting during the training process. Therefore, predictor 2 did not perform as well as the others we trained.

References

[1] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository.* 2017. URL: `http://archive.ics.uci.edu/ml`.

[2] Scikit Learn. *KNN Scikit Tool.* 2020. URL: `https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html` (visited on 12/15/2020).

[3] MATLAB. *Hyperbolic Tangent Sigmoid Transfer Function.* 2020. URL: `https://www.mathworks.com/help/deeplearning/ref/tansig.html?searchHighlight=tansig&s_tid=srchtitle` (visited on 12/12/2020).

[4] MATLAB. *Support Vector Machines for Binary Classification.* 2020. URL: `https://www.mathworks.com/help/stats/support-vector-machines-for-binary-classification.html` (visited on 12/14/2020).