CSE514 PROGRAMMING ASSIGNMENT 2

Instructor Cynthia Ma

Tri Pham

Date: 12/01/2021

I/ Introduction

This assignment aims to provide hands-on experience with the comparison of different classification models for a given problem. Specifically, seven models are being considered: k-nearest neighbors (KNN), decision tree (DT), random forest (RF), support vector machine using polynomial kernel (SVM_Polynomial), support vector machine using the radial basis function kernel (SVM_RBF), deep neural network with sigmoid activation (DNN_Sigmoid), and deep neural network with rectified linear unit activation (DNN_ReLU). The general concept of each model will be explored in later sections. For this assignment, we will be using the Breast Cancer Wisconsin (Original) dataset provided by the University of California, Irvine (UCI) Machine Learning Repository. Our goal is to find the best classification model that can determine whether a breast tumor sample is malignant or benign. The main motivation for trying out different classifiers is from the "no free lunch" theorem. In brief, no one machine learning model performs best for all possible problems. Depending on the problem, it is important to consider the trade-offs between the speed, accuracy, and complexity of different models and find one that works best for a specific problem. As mentioned above, we are considering seven different classifiers in this assignment for our task to classify whether a data sample in the Breast Cancer dataset is malignant or benign. We will consider the computational complexity (how long it takes for each classifier to obtain the best-tuned model) and validation accuracy (how accurate these tuned models classify the held-out test data) as the main factors in determining our best learning model for this problem. The remaining sections will contain our methodology, results, and discussion.

II/ Methods

There are 699 samples in the dataset, but 16 of them contain missing feature values, denoted by "?." We will deal with these 16 samples by imputing them using relevant information from other samples' feature values. Particularly, we will utilize a multivariate imputer that estimates each attribute from all others using IterativeImputer from Scikit-learn machine learning packages. We can retrieve their iterative imputation model, where each attribute is modeled as a function of the others (hence the multivariate aspect). Each attribute is imputed sequentially to allow prior imputed values to be used as part of a model in predicting subsequent attributes. The process is then repeated multiple times to ensure proper estimation of missing values. As we are splitting data randomly into a training set that contains 90% of all samples and a testing set that contains the remaining 10%, there is a possibility that some of the samples which

contain the missing values can be found in the testing set. Therefore, to avoid data leakage, we only fit the imputer on the training set and not on the entire dataset. After imputing the missing data, we will scale our data by standardizing the features by removing the mean and scaling to unit variance. Again, to avoid data leakage, we only fit the scaler on the training set. Now that the data has been preprocessed, we will perform 10-fold cross-validation on the training set. Since we are focusing on tuning our models' hyperparameters, each choice of hyperparameters essentially represents a learning model. We will then perform 10-fold cross-validation to obtain the training result for the considered model. We will apply the same procedure to other choices of hyperparameters. Then, we can compare them and pick the best choice of hyperparameters (best-tuned model) for the corresponding classifier. By the end of our program, we will have the best-tuned model for each of our 7 classifiers. We can then test these 7 tuned models with the held-out testing data to compare their validation accuracy. Their runtime is recorded mainly from their tuning process.

The packages that assist this project are Scitkit-learn, Pandas, NumPy, and Plotly. More specifically, Scikit-learn provides all the classifiers used in this assignment, along with the imputation (IterativeImputer), scaling (StandardScaler), data splitting (train_test_split and StratifiedKFold), and hyperparameters searching method (GridSearchCV). Note that we have a class imbalance dataset (there are 458 benign and 241 malignant samples), so we assume that this distribution will be similar for both the training and testing data. The default train_test_split provided by Scikit packages follows this distribution assumption, so we only need to worry about maintaining this distribution assumption when splitting the training data into folds. To do this, we use StratifiedKFold instead of normal KFold to preserve the percentage of samples for each class when splitting the folds. In addition to Scikit-learn, Pandas and NumPy are also essential packages for every machine learning project as they provide DataFrame and common matrix operations for easy data calculation and transformation. Lastly, Plotly is a visualization package that helps create interactive graphs. We use this tool to visualize our 3D plots that contain two hyperparameters and the training accuracy of a classification model.

III/ Results

In this section, we include results of hyperparameters tuning for each classifier and their validation accuracy. Additionally, we will provide a brief description of each classifier and its general strengths and weaknesses.

1. K-Nearest Neighbors

K-Nearest Neighbors is a commonly used classification method that uses the labels of k data points whose predictor variables are most similar/closest to those of a test data point to determine its label. To determine which of the k data points are most similar to the test data point, a distance measure is used.

Common approaches are Euclidean distance, Manhattan distance, and Hamming distance. The main strength of KNN is that the model is easy to interpret. It does not assume the distribution of the data, so there is no statistical complication. However, since the model itself is the training data, it has to store all the distance measurements from all training data points to a test data point. Therefore, KNN suffers from memory complexity; the more features it includes, the more likely its performance decreases.

Below are the hyperparameters tuning results for KNN. The hyperparameters considered are n_neighbors (the number of neighbors representing k) and p (distance measurement scheme). When p = 1, then we are considering Manhattan distance. For p = 2, this is equivalent to Euclidean distance. For arbitrary $p \geq 3$, this denotes Minkowski distance. We see that the highest train accuracy 97.1% is achieved when k = 21 and Euclidean distance is used.

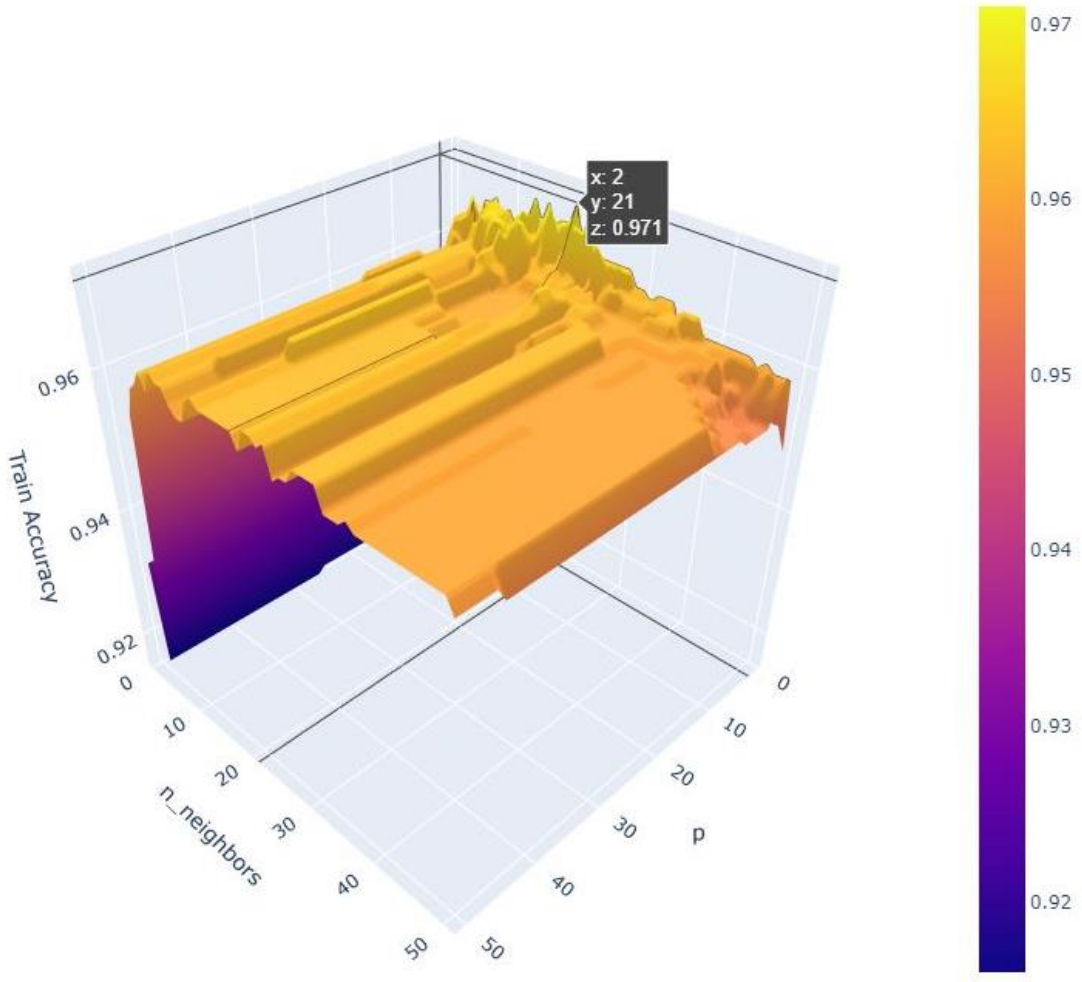

Fig1. Hyperparameters tuning results for KNN classifier

2. Decision Tree

Decision Tree is another commonly applied method in supervised learning. The process of building a DT is as follows. We first start at the root where we pick a feature to split the training data into subsets. Then, we repeat picking split points on each child node until all data points in a subset have

similar enough labels or there are no more features that can be split. The main advantage of DT is that the model can be visualized and thus are simple to understand when reading in depth-first search fashion. Moreover, DT is good at predicting classes when the boundaries between classes are non-linear. However, a big disadvantage of DT is that it tends to overfit. Furthermore, DT can be unstable because small variations in the data might result in a completely different tree being generated.

Below are the hyperparameters tuning results for DT. The hyperparameters considered are min_samples_leaf (the minimum number of samples required to be at a leaf node) and max_depth (the maximum depth of the tree). We see that the highest train accuracy 95.2% is achieved when there is a minimum number of 5 samples required to be at a leaf node and the maximum depth of the tree is at most 5 levels.

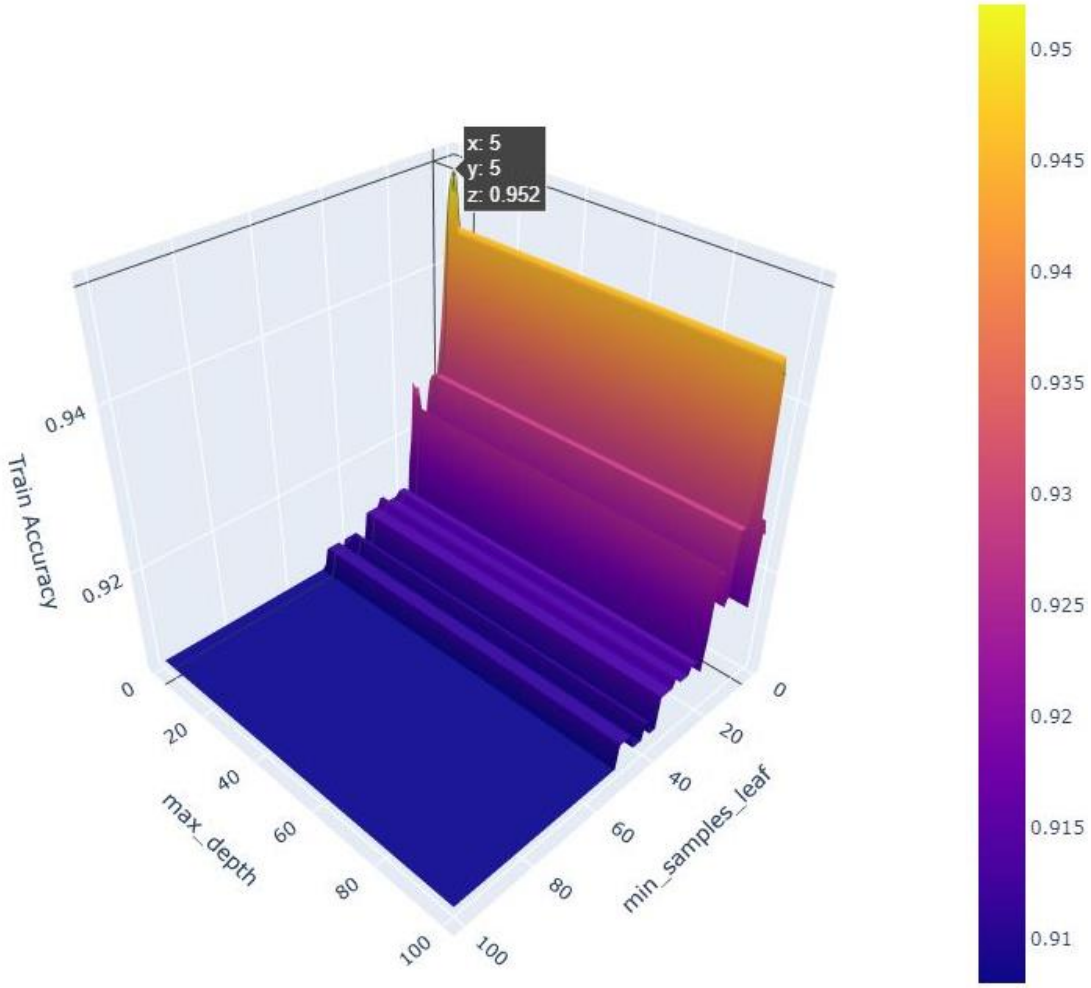

Fig2. Hyperparameters tuning results for DT classifier

3. Random Forest

Random Forest is a well-known ensemble extension for decision trees. RF considers multiple DTs through the bootstrap aggregating technique. First, we generate a new set of samples by sampling from

the training data with replacement. Then, this bootstrapped dataset is used to generate a decision tree, with the additional constraint that only a few features are selected for splitting at each node. We iteratively go through the first two steps to add more trees to our forest of DTs. Finally, to predict on a new data point, all trees are evaluated and get an equal vote in the prediction. The main advantage of RF is that the model addresses the weaknesses of DT. Since RF takes into account various DTs, it decreases the tendency to overfit. Also, RF can help find the globally optimal (or best) tree configuration. However, a drawback of RF is that it may require significant computational power and memory to derive and store the DTs. Furthermore, RF loses its interpretability since it now considers too many DTs.

Below are the hyperparameters tuning results for RF. The hyperparameters considered are max_depth (the maximum depth of the tree) and n_estimators (the number of trees in the forest). We see that the highest train accuracy 97.3% is achieved when the maximum depth of a tree is at most 6 levels and there are 200 trees in the forest.

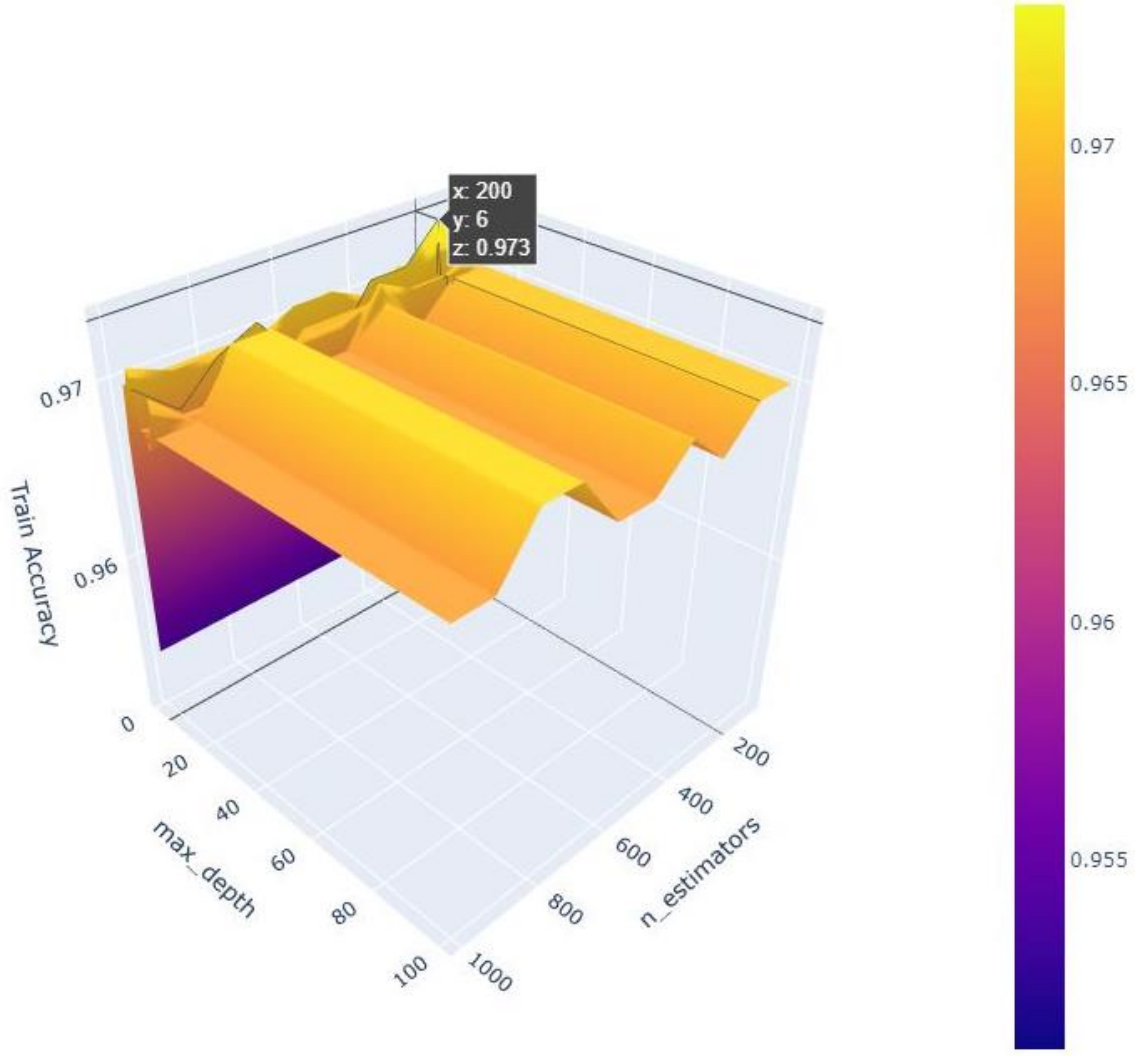

Fig3. Hyperparameters tuning result for RF classifier

4. Support Vector Machine

Support Vector Machine is mostly used in classification problems. The main idea in SVM is that we plot each data point as a point in a higher-dimensional space with the value of each attribute being the value of a particular coordinate. Then, using a hyperplane, we want to separate the two classes as best as possible. The main strength of SVM is that the model is effective in high dimensional spaces and is still effective when the number of attributes is greater than the number of samples. SVM is also versatile where different kernel functions can be used for the decision function. However, to avoid overfitting, it is important to carefully choose the kernel function as well as the regularization term.

Below are the hyperparameters tuning results for SVM_Polynomial. The hyperparameters considered are degree (degree of the polynomial kernel function) and C (regularization parameter). We see that the highest train accuracy 95.7% is achieved when the model considers third-degree polynomial and the regularization parameter is 6.9.

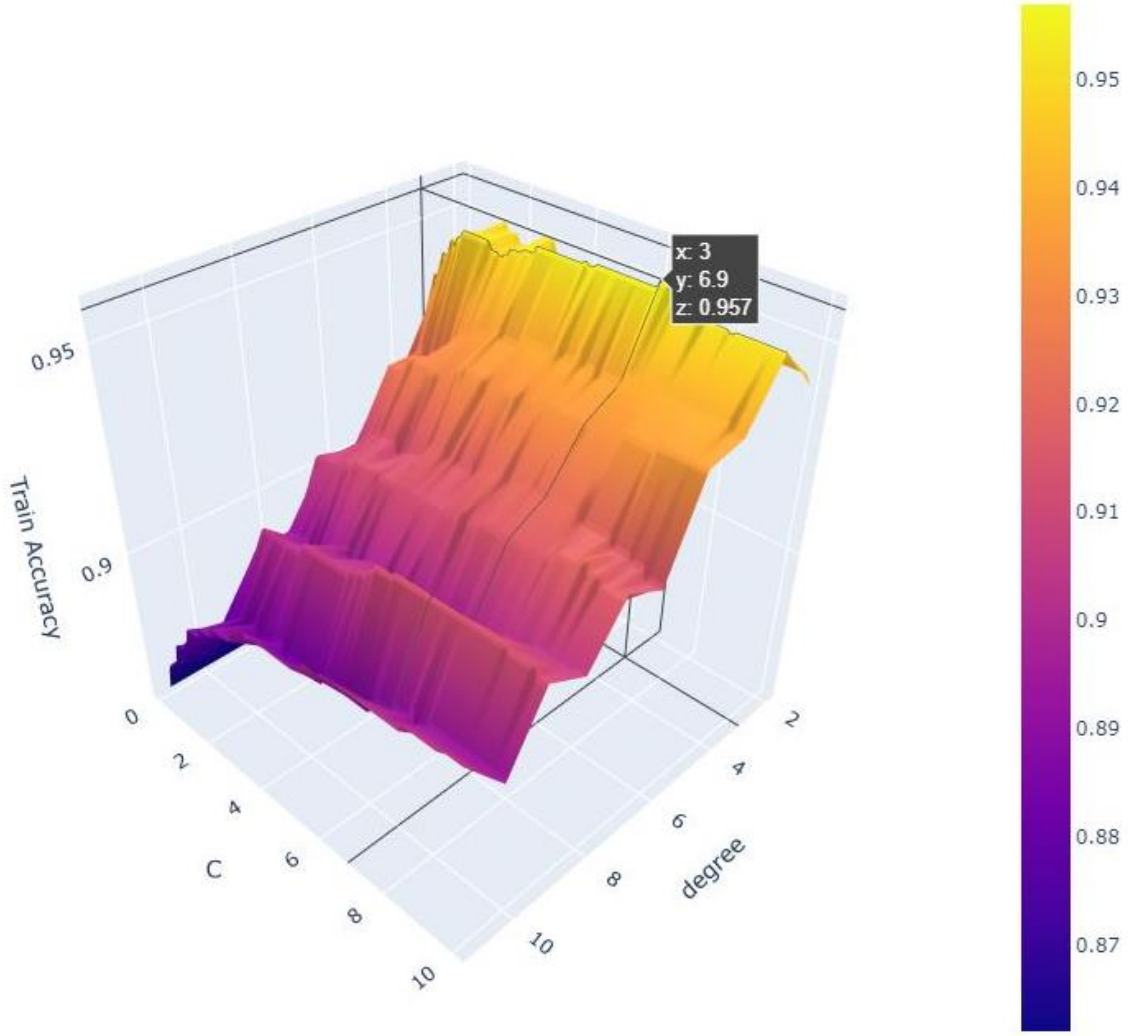

Fig4. Hyperparameters tuning result for SVM classifier using polynomial kernel

Next is the hyperparameters tuning results for SVM_RBF. The hyperparameters considered are gamma (kernel coefficient for RBF kernel) and C (regularization parameter). We see that the highest train accuracy 96.8% is achieved when the model considers C = 1.5 and gamma = 0.1.

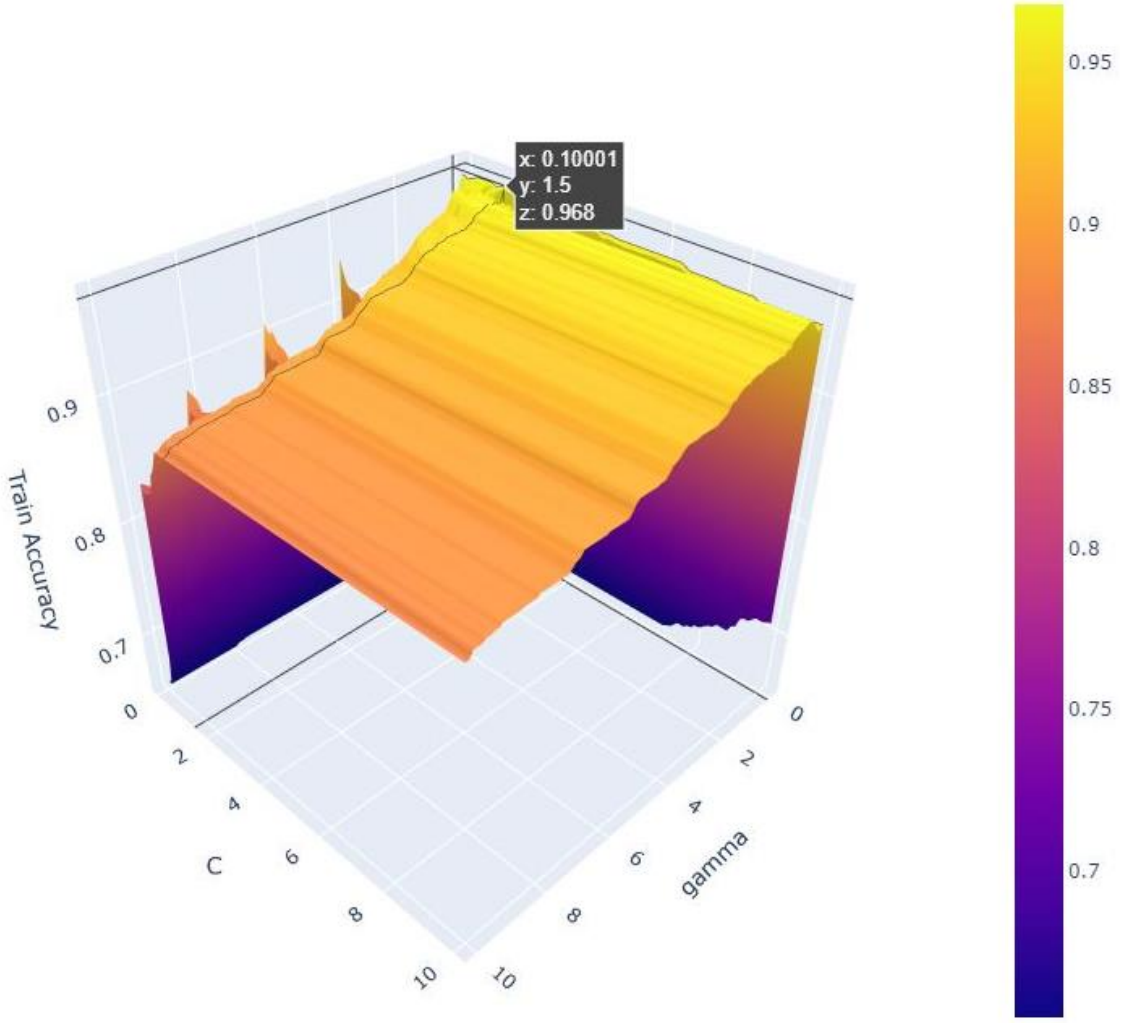

Fig5. Hyperparameters tuning result for SVM classifier using RBF kernel

5. Deep Neural Network

A Deep Neural Network is essentially an Artificial Neural Network (ANN) with multiple hidden layers. The simplest ANN is usually interpreted as a perceptron, a binary classifier that transforms an input data point's feature values with multiplicative weights, sums the results along with an additive bias, before applying a function to output the data point's classification. From this foundation, DNN is a multi-layer perceptron. It will contain many hidden layers (with a various number of nodes in each hidden layer) between the input layer and the output layer. The main advantage of DNN is its capability to learn non-linear models. DNN is also non-probabilistic, so there is no statistical complication. Nonetheless, training a neural network (hyperparameters tuning) can be time-consuming due to the complexity of the network architecture. Neural network is also susceptible to overfitting.

Below are the hyperparameters tuning results for DNN_Sigmoid. The hyperparameters considered are hidden_layer_sizes (where the i^{th} element represents the number of neurons in the i^{th} hidden layer) and learning_rate_init (this learning rate controls the step-size in updating the weights). We are fixing DNN at three hidden layers and tuning the number of hidden nodes in each layer. It might be the case that many models perform similarly well in this case. After training, we receive the best estimator whose first hidden layer contains 5 nodes, followed by 15 nodes in the second hidden layer and 15 nodes in the third layer. Also, the learning rate for this best model is 0.401.

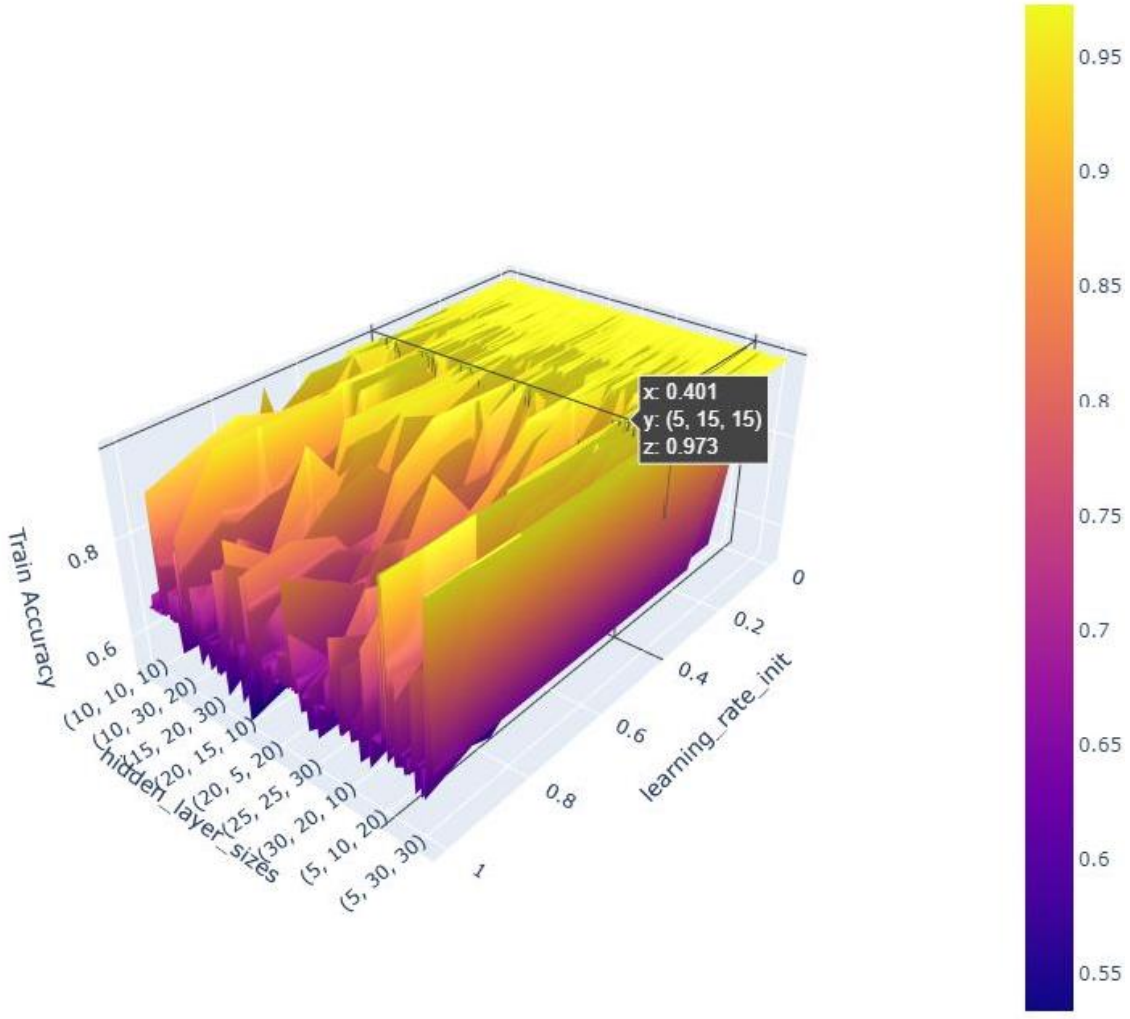

Fig6. Hyperparameters tuning result for DNN classifier using sigmoid activation

Finally, we have the hyperparameters tuning results for DNN_ReLU. The hyperparameters considered are the same as those of DNN_Sigmoid. After training, we receive the best estimator whose contains five nodes in each of its hidden layers. The learning rate for this estimator is also 0.401.

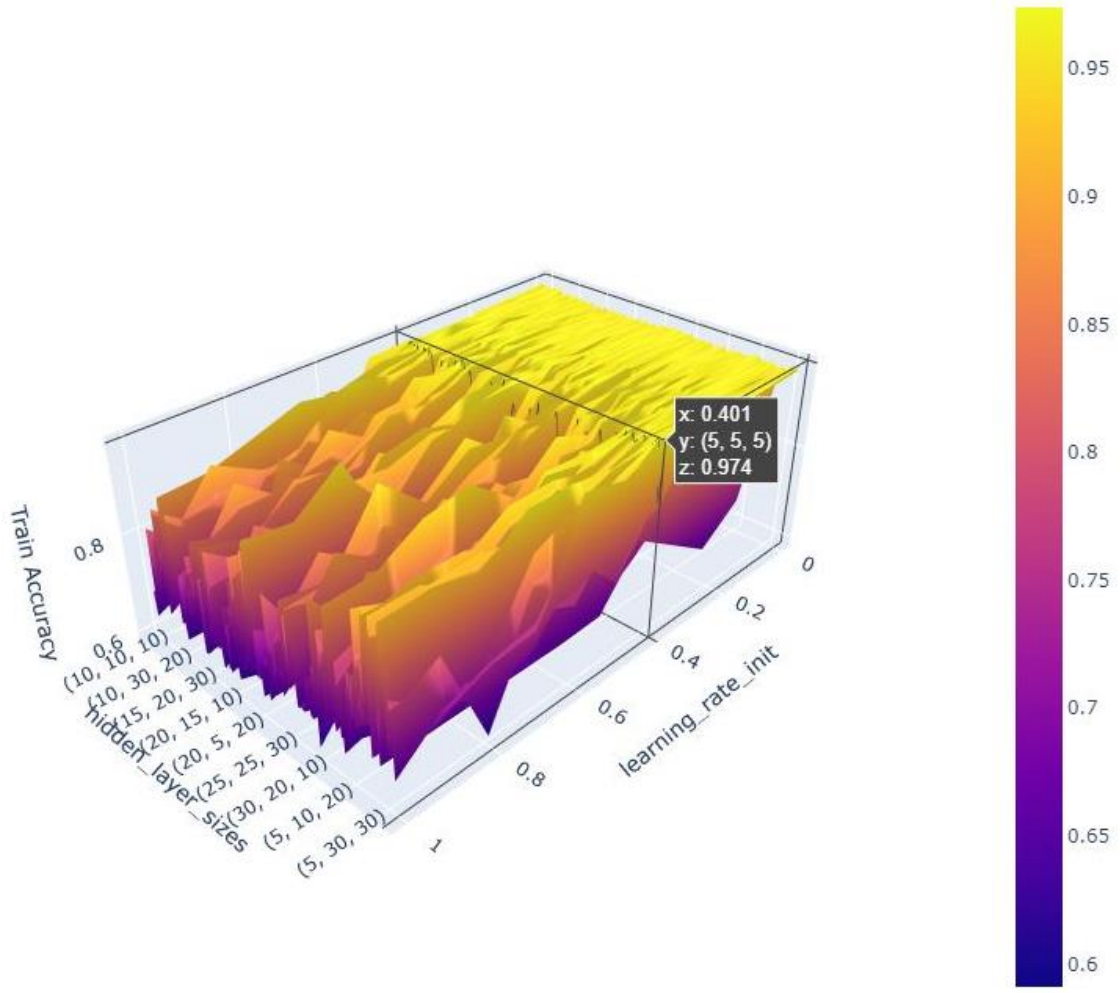

Fig7. Hyperparameters tuning result for DNN classifier using ReLU activation

IV/ Discussion

Using the (10%) held-out data for testing, we achieve the test accuracy below. With the best hyperparameters configuration for each model, KNN, both SVMs, and DNN_ReLU achieve the highest test accuracy of 97.1%, though the other models still show good results that are above 90% accuracy.

Test Accuracy	
K-Nearest Neighbors	**0.971**
Decision Tree	0.929
Random Forest	0.957
SVM using polynomial kernel	**0.971**
SVM using RBF kernel	**0.971**
DNN with sigmoid activation	0.957
DNN with ReLU activation	**0.971**

Fig8. Test accuracy for each model on the held-out data

Next, we look at the runtime of each model for their whole training process as well as the runtime for their test process.

	Time to tune each model's hyperparameters (s)	**Hyperparameters combinations**
K-Nearest Neighbors	134.22	2500
Decision Tree	14.53	10000
Random Forest	690.96	1000
SVM using polynomial kernel	**5.47**	**1000**
SVM using RBF kernel	223.78	10100
DNN with sigmoid activation	221.57	2376
DNN with ReLU activation	246.30	2376

Fig9. The runtime of each model's whole training process

Time taken for each tuned model on the final test data (s)	
K-Nearest Neighbors	0.0070
Decision Tree	**0.0**
Random Forest	0.0185
SVM using polynomial kernel	0.0005
SVM using RBF kernel	0.0005
DNN with sigmoid activation	**0.0**
DNN with ReLU activation	0.0005

Fig10. The runtime of each tuned model on the testing data

As expected for runtime, complex models such as RF or DNN took a significant amount of time to tune their hyperparameters. Furthermore, depending on the computational resources and the difference in the number of hyperparameters combinations for each classifier, some models might appear as though they have similar runtime. For example, for SVM_RBF, its runtime was achieved after 101000 fits (10100 combinations where there are 10 folds for each combination) while for DNN, the runtime is from 23760 fits (2376 combinations and there are 10 folds for each combination). It is challenging to fix a constant number of combinations across all models since each model has its hyperparameters and some appear in string form (hidden_layer_sizes), integer (max_depth, degree), float (C, gamma), et cetera. To achieve the runtime results above, we utilize the parallel computation embedded in the object instance. Specifically, we used all processors when running training for KNN, RF, and GridSearchCV since these instances are

provided with the parallelization option (n_jobs = -1). As for validating with test data, since we are using the built-in scoring function from Scikit packages, the runtime for this process is negligible across all models.

Given this problem, we choose SVM since the model has a relatively fast training/tuning time while still providing high test accuracy. Our task of classifying a data sample between only two classes aligns well with the common use of SVM is another reason why we choose SVM. As mentioned in the SVM section, since the classifier is versatile, it is important to consider different kernels and others hyperparameters. In this case, between SVM using polynomial kernel and SVM using RBF kernel, we would choose the former one simply because its recorded training runtime was much faster. However, given that the inconsistency in the number of combinations, it is also fine to use SVM with RBF kernel. Regardless, for a new dataset, we can focus more on narrowing down the combinations of hyperparameters for each classifier within a more stable interval (maybe [2000-2500]). Since this is mentioned, it is also worth trying out more kinds of hyperparameters for each classifier. A different direction from the current pipeline can also be applying Principle Component Analysis (PCA) to features and only choosing the transformed features that help contain at least 90% of the information. Since the new dataset might have more features, this approach will be useful. We can also use different feature selection methods to filter our features into a smaller subset before using them for testing.

V/ Conclusion

Overall, this project helps provide hands-on experience with a comparison of different classification models for the problem of classifying whether a breast tumor sample is malignant or benign. We discussed our pipeline from data imputation to how we trained models in different classifiers and searched for their best hyperparameter configuration. We recorded the time it took for the tuning process as well as the testing process. Finally, we discussed our results and decided that our final model choice is SVM with polynomial kernel since it satisfied our expectation with small computational complexity compared to other models and solid validation accuracy. Our code can be found at https://github.com/TriPham2110/DataMining-ModelComparison in case the user wants to reproduce the results. Any comments and feedback are also welcome.