Sheetal Sattiraju
20 November 2022

Iris Dataset Analysis

Introduction

The Iris Dataset is a very popular dataset in pattern recognition. This particular dataset contains three different classes with 50 samples each and distinct features. The classes are as follows: Iris Setosa, Iris Versicolor and Iris Virginica and the key attributes for differentiation are as follows: sepal length, sepal width, petal length and petal width (cm). The goal is to predict and accurately classify the Iris class for each flower. In this paper, I will be attempting to get to the highest accuracy possible, using four different models: SVM, RF, MLP and KNN. As we can see from Figure 1:

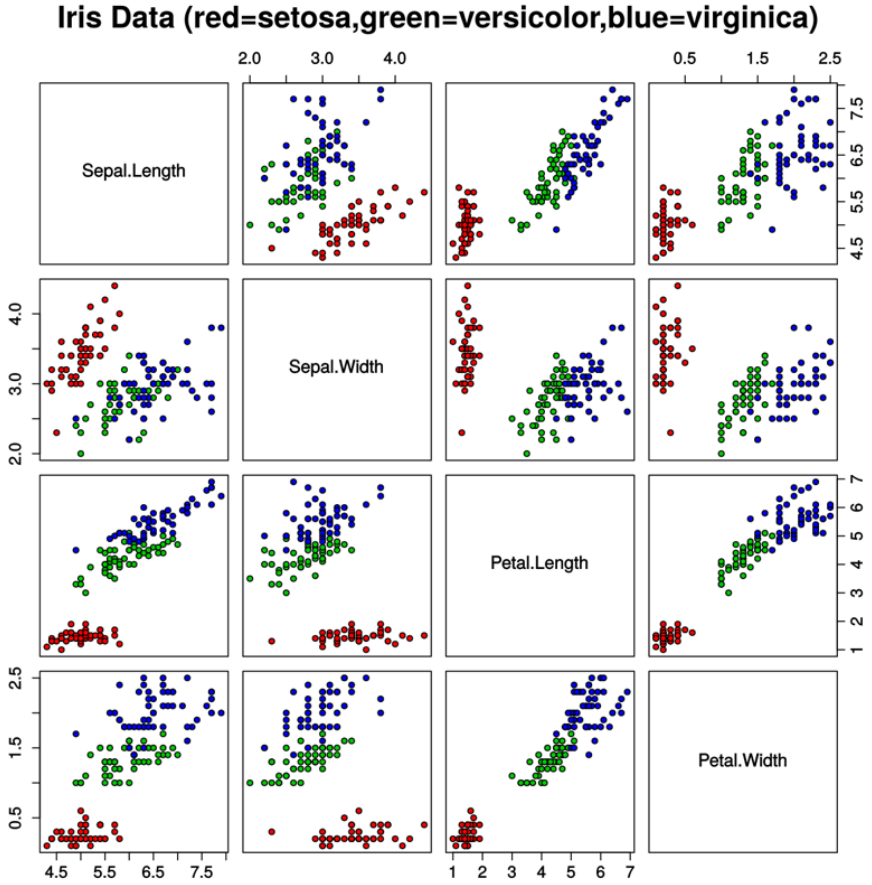

Figure 1

We can note that the Iris Setosa is linearly separable for all classes, and will likely be the easiest to classify. In addition, we can also note that the sepal width and sepal length seem to be the least separable for Iris Versicolor and Iris Virginica. We can see this further in detail with this Figure 2:

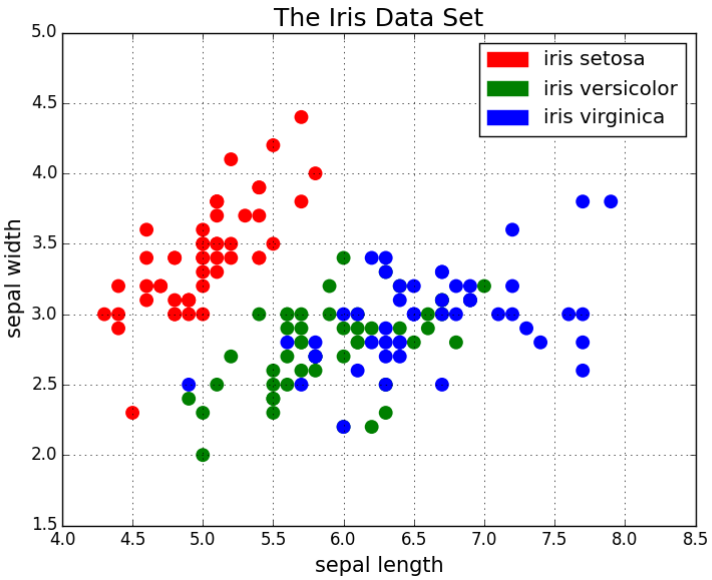

Figure 2

Experiments

I. SVM

SVM, aka Support Vector Machine, is a supervised learning algorithm used for classification problems. This method is a generalization of the maximal margin classifier, since it is meant to target the non-linearly separable classes, such as Iris Versicolor & Virginica. In the code, I make use of SVC, the support vector classifier to target these two classes specifically. SVC makes use of a soft margin and does not perfectly separate the classes, but instead focuses on the bigger picture. We can see this in Figure 3. This is key to not overfitting or incorrectly classifying the data. In the Iris Dataset, this is especially important, since Sepal.Length is tricky to classify accurately. In the case of the SVM on the Iris Dataset, the accuracy lies between 96.6667% to 100%

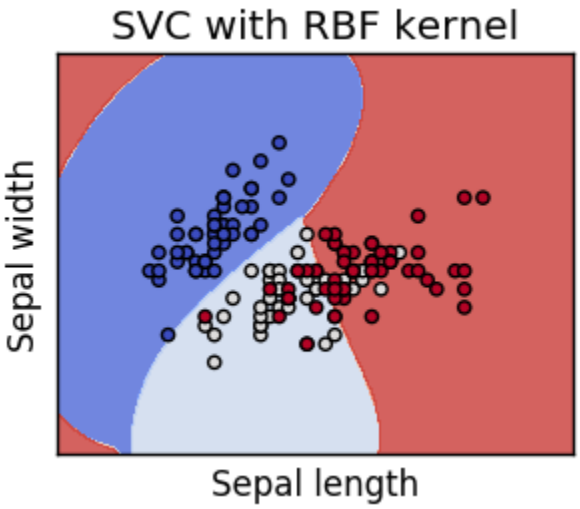

Figure 3

SVM Run 1:

```
In [7]: from sklearn.svm import SVC
        svn = SVC()
        svn.fit(X_train, y_train)

Out[7]: SVC()

In [8]: # Predict from the test dataset
        predictions = svn.predict(X_test)
        # Calculate the accuracy
        from sklearn.metrics import accuracy_score
        accuracy_score(y_test, predictions)

Out[8]: 0.9666666666666667
```

SVM Run 2:

```
        accuracy_score(y_test, predictions)

Out[8]: 1.0
```

Classification Report for SVM Run 1:

```
In [9]: from sklearn.metrics import classification_report
        print(classification_report(y_test, predictions))
```

	precision	recall	f1-score	support
0.0	1.00	1.00	1.00	11
1.0	0.90	1.00	0.95	9
2.0	1.00	0.90	0.95	10
accuracy			0.97	30
macro avg	0.97	0.97	0.96	30
weighted avg	0.97	0.97	0.97	30

Classification Report for SVM Run 2:

```
In [9]: from sklearn.metrics import classification_report
        print(classification_report(y_test, predictions))
```

	precision	recall	f1-score	support
0.0	1.00	1.00	1.00	11
1.0	1.00	1.00	1.00	7
2.0	1.00	1.00	1.00	12
accuracy			1.00	30
macro avg	1.00	1.00	1.00	30
weighted avg	1.00	1.00	1.00	30

We can note here that SVM had no trouble separating the Setosa class from Versicolor and Virginica classes, since it was easily separable. The model only had an issue, in Run 1, in

predicting for one sample, likely due to the overlap in Sepal.Length data for Versicolor and Virginica.

After using SVC, I decided to try adding GridSearch to the model. I fit the data to GridSearch and printed a new classification report:

Classification Report for SVM Run 1 with GridSearch:

```
In [15]: print(classification_report(y_test, pred_grid))
```

	precision	recall	f1-score	support
0.0	1.00	1.00	1.00	11
1.0	0.90	1.00	0.95	9
2.0	1.00	0.90	0.95	10
accuracy			0.97	30
macro avg	0.97	0.97	0.96	30
weighted avg	0.97	0.97	0.97	30

```
In [16]: accuracy_score(y_test, pred_grid)
Out[16]: 0.9666666666666667
```

Classification Report for SVM Run 2 with GridSearch:

```
In [15]: print(classification_report(y_test, pred_grid))
```

	precision	recall	f1-score	support
0.0	1.00	1.00	1.00	11
1.0	1.00	1.00	1.00	7
2.0	1.00	1.00	1.00	12
accuracy			1.00	30
macro avg	1.00	1.00	1.00	30
weighted avg	1.00	1.00	1.00	30

```
In [16]: accuracy_score(y_test, pred_grid)
Out[16]: 1.0
```

We can note here that adding GridSearch to this model brought the accuracy to around the same and did not help the model increase its accuracy. (GridSearch is a turning technique that attempts to find the most optimal hyperparameters of a model)

See 'SVM On Iris Dataset' to see all of the code used to test this model.

II. Random Forest Classifier

Random Forest Classifier is a supervised algorithm and ensemble learning method that consists of a large number of individual decision trees, which individually indicates a prediction (the class) and decides the class based on a majority vote of these trees. The predictive nature of this algorithm allows for accurate classification, while the randomness prevents overfitting of the

data. In the case of the Iris Dataset, the accuracy lied around 93.333%-100%. In Figure 3, we can see how RandomForest separates and classifies the data based on the key attributes: Sepal Width, Length, Petal Width and Petal Length.

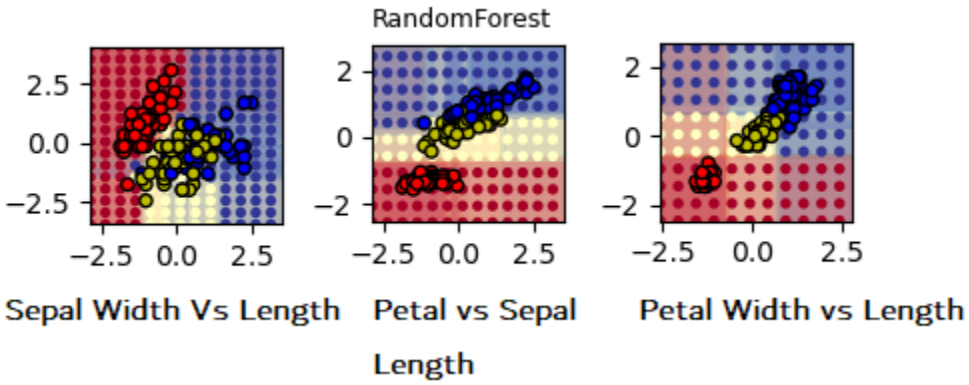

Figure 4

RF Run 1:

```
In [9]: from sklearn.ensemble import RandomForestClassifier
        rf = RandomForestClassifier(n_estimators=100)
        rf.fit(X_train, y_train)

Out[9]: RandomForestClassifier()

In [10]: from sklearn.metrics import accuracy_score, classification_report
         predictions = rf.predict(X_test)

In [11]: accuracy_score(y_test, predictions)

Out[11]: 0.9666666666666667
```

RF Run 2:

```
In [8]: accuracy_score(y_test, predictions)

Out[8]: 0.9
```

RF Run 3:

```
In [8]: accuracy_score(y_test, predictions)

Out[8]: 0.9333333333333333
```

RF Run 4:

```
In [8]: accuracy_score(y_test, predictions)

Out[8]: 1.0
```

Classification Report for RF Run 1:

```
In [12]: print(classification_report(y_test, predictions))
```

```
                  precision    recall  f1-score   support

             0.0       1.00      1.00      1.00        10
             1.0       1.00      0.91      0.95        11
             2.0       0.90      1.00      0.95         9

        accuracy                           0.97        30
       macro avg       0.97      0.97      0.97        30
    weighted avg       0.97      0.97      0.97        30
```

We can note, once again, that this model had no trouble separating the Setosa class from Versicolor and Virginica classes. However, Random Forest Classifier has a lower initial precision for the Versicolor and Virginica class, than the SVM classifier. This classifier came up with an accuracy between 90% - 100%, but is still viable in classifying the Iris Dataset accurately. It is important to note that the accuracy of this model is not as consistent as SVM, which makes SVM the better model in classifying this particular dataset.

After using this classifier, I decided to try adding GridSearch to the model. However, I used a slightly different approach for GridSearch than for SVM. I used a KFold of 5:

Classification Report for RF Run 1 with GridSearch:

```
In [14]: from sklearn.model_selection import GridSearchCV
         from sklearn.model_selection import KFold
         rf_grid = {'max_samples': [0.1, 0.2, 0.3, 0.4],
                    'max_features': [1, 2],
                    'n_estimators':[10, 50, 100],
                    'max_depth': [8, 9, 10]
                    }
         rf_k = KFold(n_splits=5)
         rf = GridSearchCV(RandomForestClassifier(), rf_grid, cv=rf_k)
         rf.fit(X_train, y_train)
```

```
Out[14]: GridSearchCV(cv=KFold(n_splits=5, random_state=None, shuffle=False),
                      estimator=RandomForestClassifier(),
                      param_grid={'max_depth': [8, 9, 10], 'max_features': [1, 2],
                                  'max_samples': [0.1, 0.2, 0.3, 0.4],
                                  'n_estimators': [10, 50, 100]})
```

```
In [15]: predictions2 = rf.predict(X_test)
         print(classification_report(y_test, predictions2))
```

```
                  precision    recall  f1-score   support

             0.0       1.00      1.00      1.00        10
             1.0       1.00      1.00      1.00        11
             2.0       1.00      1.00      1.00         9

        accuracy                           1.00        30
       macro avg       1.00      1.00      1.00        30
    weighted avg       1.00      1.00      1.00        30
```

```
In [16]: accuracy_score(y_test, predictions2)
```

```
Out[16]: 1.0
```

Accuracy for RF Run 2 with GridSearch:

```
In [13]: accuracy_score(y_test, predictions2)
Out[13]: 0.9333333333333333
```

Once again, we can note that the GridSearch tuning did improve the accuracy rate of the Iris Dataset from 96.666% to 100% in the first run. Accordingly, for the second run, the original model's accuracy improved from 90% to 93.333% with GridSearch, proving successful in both instances.
See 'RF On Iris Dataset' to see all of the code used to test this model.

III. MLP

MLP, aka Multilayer perceptron, is a neural network deep learning algorithm, in which the mapping between inputs and outputs is non-linear. MLP makes use of hidden layers and takes a set of inputs to provide an output using an activation function. The feedforward nature of this algorithm is from each layer "feeding" into the next layer, until the output is found. We can see this process in Figure 5, below. For this dataset, the Iris Dataset, surprisingly, performed well and achieved a 97.777%, even after further runs.

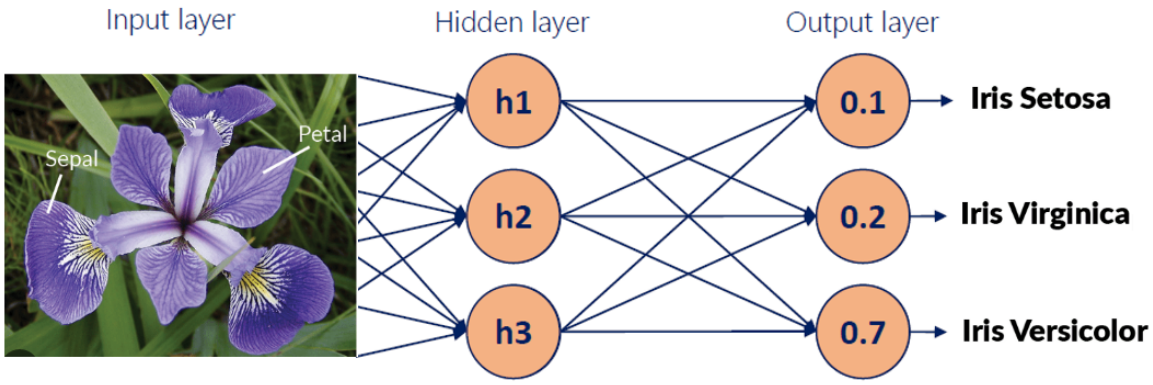

Figure 5

MLP Run 1 & Classification Report:

```
In [7]: mlp=MLPClassifier(hidden_layer_sizes=(100,), activation='relu', solver='adam', alpha=0.0001, max_iter=800)

In [8]: mlp.fit(X_train, y_train)
        predictions = mlp.predict(X_test)

In [9]: accuracy_score(y_test, predictions)

Out[9]: 0.9777777777777777

In [10]: print(classification_report(y_test, predictions))
```

	precision	recall	f1-score	support
0.0	1.00	1.00	1.00	16
1.0	1.00	0.94	0.97	18
2.0	0.92	1.00	0.96	11
accuracy			0.98	45
macro avg	0.97	0.98	0.98	45
weighted avg	0.98	0.98	0.98	45

We can note that this specific algorithm only struggled for one sample regarding the final class, Iris Virginica. Besides that, the algorithm works pretty fast, and efficiently for such a high accuracy. I did not run GridSearch on this classifier to avoid the possibility of overfitting.
See 'MLP On Iris Dataset' to see all of the code used to test this model.

IV. KNN

KNN, or k-Nearest Neighbor, is a supervised learning classifier that, given a labeled training set, uses proximities of a class type to predict what a datapoints class may be. The 'k' in KNN is the chosen number of nearest neighbors used to determine what a class is. For instance, if k = 5, then KNN will take the 5 closest points to the datapoint we are at and take the average of these to assign a class to that datapoint. We can see this process in Figure 6, below:

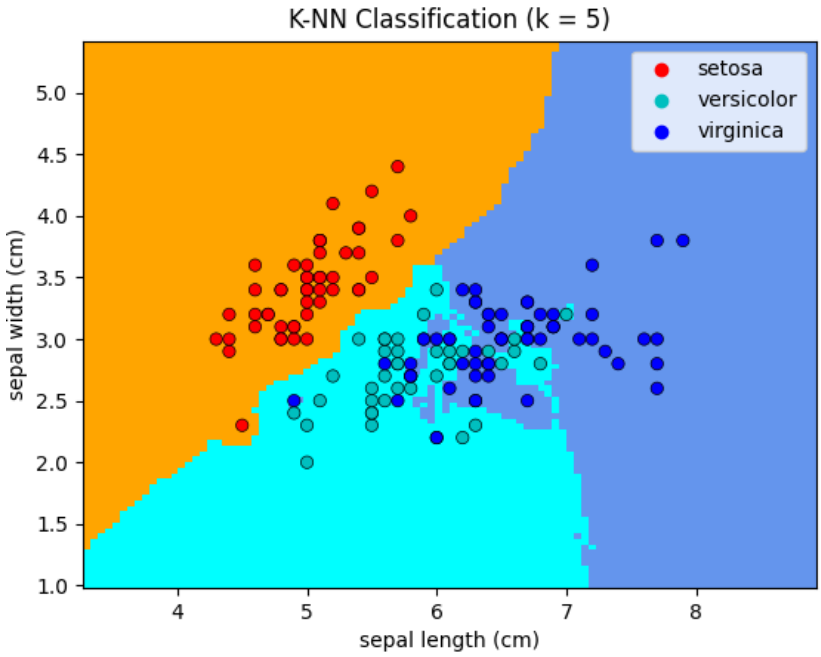

Figure 6

For this dataset, I used K values 3, 5 and 7 and got accuracy rates of 93.333%-100% on the iris dataset. I additionally tested with K values of 1, 9, which are not included in this document:

KNN Run 1 & Classification Report

```
In [19]: from sklearn.neighbors import KNeighborsClassifier
         knn = KNeighborsClassifier(7)
         knn.fit(X_train, y_train)

Out[19]: KNeighborsClassifier(n_neighbors=7)

In [20]: # Predict from the test dataset
         predictions = knn.predict(X_test)
         # Calculate the accuracy
         from sklearn.metrics import accuracy_score
         accuracy_score(y_test, predictions)

Out[20]: 0.9666666666666667

In [21]: from sklearn.metrics import classification_report
         print(classification_report(y_test, predictions))

                       precision    recall  f1-score   support

                  0.0       1.00      1.00      1.00         5
                  1.0       0.93      1.00      0.96        13
                  2.0       1.00      0.92      0.96        12

             accuracy                           0.97        30
            macro avg       0.98      0.97      0.97        30
         weighted avg       0.97      0.97      0.97        30
```

KNN Run 2 & Classification Report

```
In [7]: from sklearn.neighbors import KNeighborsClassifier
        knn = KNeighborsClassifier(5)
        knn.fit(X_train, y_train)

Out[7]: KNeighborsClassifier()

In [8]: # Predict from the test dataset
        predictions = knn.predict(X_test)
        # Calculate the accuracy
        from sklearn.metrics import accuracy_score
        accuracy_score(y_test, predictions)

Out[8]: 0.9333333333333333
```

KNN Run 3 & Classification Report:

```
In [7]: from sklearn.neighbors import KNeighborsClassifier
        knn = KNeighborsClassifier(3)
        knn.fit(X_train, y_train)

Out[7]: KNeighborsClassifier(n_neighbors=3)

In [8]: # Predict from the test dataset
        predictions = knn.predict(X_test)
        # Calculate the accuracy
        from sklearn.metrics import accuracy_score
        accuracy_score(y_test, predictions)

Out[8]: 1.0

In [9]: from sklearn.metrics import classification_report
        print(classification_report(y_test, predictions))
```

	precision	recall	f1-score	support
0.0	1.00	1.00	1.00	9
1.0	1.00	1.00	1.00	9
2.0	1.00	1.00	1.00	12
accuracy			1.00	30
macro avg	1.00	1.00	1.00	30
weighted avg	1.00	1.00	1.00	30

We can note that KNN, for this iris dataset, had the most accurate classification with k=3, however, after additional experiments, I have recognized that as long as the k is between 1-9, KNN classification gives an accuracy in the 93-100% range regardless. We can see from Run 1 that one sample in the Versicolor class was labeled incorrectly.
See 'KNN On Iris Dataset' to see all of the code used to test this model.

Analysis of Experiments

From these four experiments, we can see that all four methods can be used to deliver a decently accurate model. Almost all the models have an accuracy rate above 93.333%, with the exception of one instance of a RandomForestClassifier Run, but most of the runs for this model were above this rate as well. MLP had the most consistent accuracy, at 97.77%, but all of the models were able to reach 100% in some test runs. SVM and KNN had almost identical results at 93-96% most of the time, while RandomForest had more varying accuracies between 90-100%. None of the models had an overfitting issue and seemed sustainable. I used GridSearch for SVM and RandomForest, and had results in which turning the hyperparameters either helped the model reach a slightly higher accuracy or no change at all.

We can see with all of the classifiers we have tested today that they all either mislabel 1-3 of the Iris Versicolor or Virginica classes. None of these classifiers had trouble labeling the Setosa, as predicted at the beginning of this paper. *Figure 1* and *Figure 2* showcase the overlapping data points between sepal width and sepal length. Each model had its own way of targeting this non-linearly separable data. In *Figure 3,* we can see how the SVC (kernel RBF) uses a rounded line or rounded shape to classify the data points accordingly. In *Figure 4,* we can see how the RandomForestClassifier's decision trees sit in contrasting regions and classify the points that fall in that region. *Figure 5* showcases MLP's feedforward algorithm; we can see how

the neural network takes in the input layer, sends it to the hidden layers and uses an activation function to create probabilities and assign a class from this information. *Figure 6* accurately shows how the KNN algorithm looks when applied to the iris data set for k=5. We can see how KNN managed to label most of the Versicolor and Virginica accurately, even though they overlap significantly (the light blue vs dark blue shaded regions).

Conclusion and Future Works

In short, I experimented with SVM, RF, MLP and KNN due to their abilities to target non-linearly separable classes. All four of these models are highly accurate and successful. In the future, I would likely try SVM with a different kernel (such as linear, polynomial) to see if that produces less or more accurate results. I would also try other optimizer techniques besides GridSearch and see if that would improve the results for this dataset. Other possible models for testing would be CNN, NLP, other ensemble learning methods, and etc. As long as the model can separate non-linear classes, it should result in a high accuracy.

Citations

- For GridSearch (SVM): https://www.kaggle.com/code/spscientist/iris-classification-using-svm-and-gridsearch/notebook
- For GridSearch (RF): https://www.embedded-robotics.com/iris-dataset-classification/#data-classification-using-random-forest-classifier

For Graphs:

- Figure 2: https://www.pybloggers.com/2015/09/my-first-time-using-matplotlib/
- Figure 3: https://scikit-learn.org/0.18/auto_examples/svm/plot_iris.html
- Figure 4: https://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_iris.html
- Figure 6: Modified from: https://scikit-learn.org/stable/auto_examples/neighbors/plot_classification.html