Model Design and Selection with Scikit-learn

Tuning, Training, and Evaluating Models with Scikit-learn

Frank Ceballos
Jul 8, 2019 · 11 min read ★

Photo by Alex wong on Unsplash

Purpose: The purpose of this article is to build a pipeline from start to finish in order to access the predictive performance of 18 machine learning models on a synthetic data set.

Materials and methods: Using Scikit-learn, we generate a Madelon-like data set for a score normalization. (3) A feature selection algorithm is applied to reduce the number of features. (4) A machine learning algorithm is trained and evaluated. The predictive power of the 18 trained classifiers will be evaluated using the area under the receiver operating curve (AUC).

Hardware: We train and evaluate our models on a workstation equipped with Inter(R)Core(TM) i7–8700 with 12 CPU @ 3.70 Ghz and NVIDIA GeForce RTX 2080.

Note: In the case you're starting from scratch, I will advise you to follow this article and install all the necessary libraries. You're welcome to fork my repository that contains the entire contents of this article.

Data Set

We will generate a Madelon-like synthetic data set using Scikit-learn for a classification task. The Madelon data set is an artificial data set that contains 32 clusters placed on the vertices of a five-dimensional hyper-cube with sides of length 1. The clusters are randomly labeled 1 or -1 (2 classes).

The data set that we will generate will contain 30 features, where 5 of them will be informative, 15 will be redundant (but informative), 5 of them will be repeated, and the last 5 will be useless since they will be filled with random noise. The columns of the data set will be ordered as follows:

1. **Informative features — Columns 1–5**: These features are the only features you really need to built your model. Hence, a five-dimensional hyper-cube.
2. **Redundant features — Columns 6–20:** These features are made by linearly combining the informative features with different random weights. You can think of these as engineered features.
3. **Repeated features — Columns 21–25**: These features are drawn randomly from either the informative or redundant features.
4. **Useless features — Columns 26–30:** These features are filled with random noise.

```python
#                          1. Importing Libraries                             #
###############################################################################
# For reading, visualizing, and preprocessing data
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.feature_selection import RFE, RFECV
from sklearn.model_selection import train_test_split, GridSearchCV, KFold
from sklearn.preprocessing import StandardScaler, Imputer
from sklearn.pipeline import Pipeline
from sklearn import metrics

# Classifiers
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis, QuadraticDiscrimi
from sklearn.ensemble import AdaBoostClassifier, BaggingClassifier, ExtraTreesClassifie
from sklearn.linear_model import RidgeClassifier, SGDClassifier
from sklearn.naive_bayes import BernoulliNB, GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.svm import LinearSVC, NuSVC, SVC
from sklearn.tree import DecisionTreeClassifier, ExtraTreeClassifier
```

01ModelDesign&Selection.py hosted with ❤ by **GitHub** view raw

Script 1 — Importing the libraries.

Now we can generate our data set.

```python
###############################################################################
#                                 2. Get data                                 #
###############################################################################
X, y = make_classification(n_samples = 1000, n_features = 30, n_informative = 5,
                           n_redundant = 15, n_repeated = 5,
                           n_clusters_per_class = 2, class_sep = 0.5,
                           random_state = 1000, shuffle = False)

# Numpy array to pandas dataframe
labels = [f"Feature {ii+1}" for ii in range(X.shape[1])]
X = pd.DataFrame(X, columns = labels)
y = pd.DataFrame(y, columns = ["Target"])
```

02ModelDesign&Selection.py hosted with ❤ by **GitHub** view raw

By randomly sampling without replacement, we create our training and test sets.

```python
################################################################################
#                          3. Create train and test set                        #
################################################################################
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20,
                                                       random_state = 1000)
```

03ModelDesign&Selection.py hosted with ❤ by GitHub view raw

Script 3 — Creating the training and test set. The size of the test set is set to be 20% of the data.

Classifiers

We will train and tune 18 classifiers and evaluate their performance using the area under the receiver operating curve (AUC). It's way beyond the scope of this article to discuss any of the technicalities of each classifier; however, for the interested reader, you can follow the links in the list shown below. Each classifier is given a label denoted by the string contained between parentheses.

1. Linear Discrimination Analysis (LDA)
2. Quadratic Discriminant Analysis (QDA)
3. Adaboost Classifier (AdaBoost)
4. Bagging Classifier (Bagging)
5. Extra Trees Classifier (Extra Trees Ensemble)
6. Gradient Boosting Classifier (Gradient Boosting)
7. Random Forest Classifier (Random Forest)
8. Ridge Classifier (Ridge)
9. SGD Classifier (SGD)
10. Bernoulli NB Classifier (BNB)

12. K Nearest Neighbors Classifier (KNN)

13. MLP Classifier (MLP)

14. Linear SVC (LSVC)

15. Nu SVC (NuSVC)

16. SVC (SVC)

17. Decision Tree Classifier (DTC)

18. Extra Tree Classifier (ETC)

```python
###############################################################################
#                               4. Classifiers                               #
###############################################################################
# Create list of tuples with classifier label and classifier object
classifiers = {}
classifiers.update({"LDA": LinearDiscriminantAnalysis()})
classifiers.update({"QDA": QuadraticDiscriminantAnalysis()})
classifiers.update({"AdaBoost": AdaBoostClassifier()})
classifiers.update({"Bagging": BaggingClassifier()})
classifiers.update({"Extra Trees Ensemble": ExtraTreesClassifier()})
classifiers.update({"Gradient Boosting": GradientBoostingClassifier()})
classifiers.update({"Random Forest": RandomForestClassifier()})
classifiers.update({"Ridge": RidgeClassifier()})
classifiers.update({"SGD": SGDClassifier()})
classifiers.update({"BNB": BernoulliNB()})
classifiers.update({"GNB": GaussianNB()})
classifiers.update({"KNN": KNeighborsClassifier()})
classifiers.update({"MLP": MLPClassifier()})
classifiers.update({"LSVC": LinearSVC()})
classifiers.update({"NuSVC": NuSVC()})
classifiers.update({"SVC": SVC()})
classifiers.update({"DTC": DecisionTreeClassifier()})
classifiers.update({"ETC": ExtraTreeClassifier()})

# Create dict of decision function labels
DECISION_FUNCTIONS = {"Ridge", "SGD", "LSVC", "NuSVC", "SVC"}

# Create dict for classifiers with feature_importances_ attribute
FEATURE_IMPORTANCE = {"Gradient Boosting", "Extra Trees Ensemble", "Random Forest"}
```

Classifier Hyper-parameters

Here we will create a dictionary whose key-value pairs consist of

- **key**: String denoting the classifier
- **value**: Dictionary of hyper-parameters for corresponding classifier

By no means do the hyper-parameters used here represent the optimal hyper-parameter grid for each classifier. You're welcome to change the hyper-parameter grid as you wish.

```
###############################################################################
#                              5. Hyper-parameters                            #
###############################################################################
# Initiate parameter grid
parameters = {}

# Update dict with LDA
parameters.update({"LDA": {"classifier__solver": ["svd"],
                                         }})

# Update dict with QDA
parameters.update({"QDA": {"classifier__reg_param":[0.01*ii for ii in range(0, 101)],
                                         }})
# Update dict with AdaBoost
parameters.update({"AdaBoost": {
                                "classifier__base_estimator": [DecisionTreeClassifier(
                                "classifier__n_estimators": [200],
                                "classifier__learning_rate": [0.001, 0.01, 0.05, 0.1,
                                 }})

# Update dict with Bagging
parameters.update({"Bagging": {
                                "classifier__base_estimator": [DecisionTreeClassifier(
                                "classifier__n_estimators": [200],
                                "classifier__max_features": [0.2, 0.3, 0.4, 0.5, 0.6,
                                "classifier__n_jobs": [-1]
                                }})

# Update dict with Gradient Boosting
parameters.update({"Gradient Boosting": {
                                        "classifier__learning_rate":[0.15,0.1,0.05,0.0
```

Read more stories this month when you create a free Medium account.

```
                                          "classifier__min_samples_leaf": [0.005, 0.01,
                                          "classifier__max_features": ["auto", "sqrt", '
                                          "classifier__subsample": [0.8, 0.9, 1]
                                           }})


# Update dict with Extra Trees
parameters.update({"Extra Trees Ensemble": {
                                              "classifier__n_estimators": [200],
                                              "classifier__class_weight": [None, "balanc
                                              "classifier__max_features": ["auto", "sqrt
                                              "classifier__max_depth" : [3, 4, 5, 6, 7,
                                              "classifier__min_samples_split": [0.005, 0
                                              "classifier__min_samples_leaf": [0.005, 0.
                                              "classifier__criterion" :["gini", "entropy
                                              "classifier__n_jobs": [-1]
                                               }})


# Update dict with Random Forest Parameters
parameters.update({"Random Forest": {
                                    "classifier__n_estimators": [200],
                                    "classifier__class_weight": [None, "balanced"],
                                    "classifier__max_features": ["auto", "sqrt", "log2
                                    "classifier__max_depth" : [3, 4, 5, 6, 7, 8],
                                    "classifier__min_samples_split": [0.005, 0.01, 0.0
                                    "classifier__min_samples_leaf": [0.005, 0.01, 0.05
                                    "classifier__criterion" :["gini", "entropy"]     ,
                                    "classifier__n_jobs": [-1]
                                     }})

# Update dict with Ridge
parameters.update({"Ridge": {
                            "classifier__alpha": [1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2,
                             }})

# Update dict with SGD Classifier
parameters.update({"SGD": {
                            "classifier__alpha": [1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2,
                            "classifier__penalty": ["l1", "l2"],
                            "classifier__n_jobs": [-1]
                             }})


# Update dict with BernoulliNB Classifier

# Update dict with GaussianNB Classifier
parameters.update({"GNB": {
                              "classifier__var_smoothing": [1e-9, 1e-8,1e-7, 1e-6, 1e-5]
                               }})

# Update dict with K Nearest Neighbors Classifier
parameters.update({"KNN": {
                              "classifier__n_neighbors": list(range(1,31)),
                              "classifier__p": [1, 2, 3, 4, 5],
                              "classifier__leaf_size": [5, 10, 15, 20, 25, 30, 35, 40, 4
                              "classifier__n_jobs": [-1]
                               }})

# Update dict with MLPClassifier
parameters.update({"MLP": {
                              "classifier__hidden_layer_sizes": [(5), (10), (5,5), (10,1
                              "classifier__activation": ["identity", "logistic", "tanh",
                              "classifier__learning_rate": ["constant", "invscaling", "a
                              "classifier__max_iter": [100, 200, 300, 500, 1000, 2000],
                              "classifier__alpha": list(10.0 ** -np.arange(1, 10)),
                               }})

parameters.update({"LSVC": {
                              "classifier__penalty": ["l2"],
                              "classifier__C": [0.0001, 0.001, 0.01, 0.1, 1.0, 10, 100]
                               }})

parameters.update({"NuSVC": {
                              "classifier__nu": [0.25, 0.50, 0.75],
                              "classifier__kernel": ["linear", "rbf", "poly"],
                              "classifier__degree": [1,2,3,4,5,6],
                               }})

parameters.update({"SVC": {
                              "classifier__kernel": ["linear", "rbf", "poly"],
                              "classifier__gamma": ["auto"],
                              "classifier__C": [0.1, 0.5, 1, 5, 10, 50, 100],
                              "classifier__degree": [1, 2, 3, 4, 5, 6]
                               }})


# Update dict with Decision Tree Classifier
parameters.update({"DTC": {
                              "classifier__criterion" :["gini", "entropy"],
                               "classifier__max_features" : ["auto", "sqrt", "log2"],
                              "classifier__max_depth" : [1,2,3, 4, 5, 6, 7, 8],
                              "classifier__min_samples_split": [0.005, 0.01, 0.05, 0.10]
                              "classifier__min_samples_leaf": [0.005, 0.01, 0.05, 0.10],
                               }})

# Update dict with Extra Tree Classifier
parameters.update({"ETC": {
                              "classifier__criterion" :["gini", "entropy"],
                              "classifier__splitter": ["best", "random"],
                              "classifier__class_weight": [None, "balanced"],
                              "classifier__max_features": ["auto", "sqrt", "log2"],
                              "classifier__max_depth" : [1,2,3, 4, 5, 6, 7, 8],
                              "classifier__min_samples_split": [0.005, 0.01, 0.05, 0.10]
                              "classifier__min_samples_leaf": [0.005, 0.01, 0.05, 0.10],
                               }})
```

Feature Selection Methods

Machine learning can involve problems with thousands of features for each training instance. Determining the optimal subset of features from a large cohort is a common task in machine learning. The advantages that one gains by doing so are numerous. For example, finding the most descriptive features reduces a model's complexity, makes it easier to find the best solution, and most importantly, it decreases the time it takes to train the model. In some instances, a slight performance boost can be gained.

Fortunately, it is often possible to considerably reduce the number of features using well established methods. However, it must be noted that by removing features your system might perform slightly worse (since you're trying to make a prediction with less information).

There are three common methods for selecting features. Namely, filter, wrapper, and embedded methods. It's beyond the scope of this article to explain them fully. Therefore, if you're not familiar with these methods, I will advise you to read this article,and this one too.

In our workflow, we will first apply a filter method to rapidly reduce the number of

1. Filter Method: Correlation-Based Feature Selection

Let's assume that if two features or more are highly correlated, we can randomly select one of them and discard the rest without losing any information. To measure the correlation between features, we will use Spearman's correlation coefficient. If two features have a Spearman's correlation value of 1, it means that they are perfectly correlated, 0 not correlated, and -1 highly correlated but in the opposite direction (one feature increases while the other decreases).

In this step of the feature selection algorithm, we first compute the absolute value of the coefficient matrix using all the features, see **Figure 1**. We then determine a group of features that have a correlation coefficient greater than 0.95. From each group of correlated features, we will select one of them and discard the rest. You're welcome to change this threshold I arbitrarily set.

```
1   ##############################################################################
2   #              6. Feature Selection: Removing highly correlated features     #
3   ##############################################################################
4   # Filter Method: Spearman's Cross Correlation > 0.95
5   # Make correlation matrix
6   corr_matrix = X_train.corr(method = "spearman").abs()
7
8   # Draw the heatmap
9   sns.set(font_scale = 1.0)
10  f, ax = plt.subplots(figsize=(11, 9))
11  sns.heatmap(corr_matrix, cmap= "YlGnBu", square=True, ax = ax)
12  f.tight_layout()
13  plt.savefig("correlation_matrix.png", dpi = 1080)
14
15  # Select upper triangle of matrix
16  upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k = 1).astype(np.bool))
17
18  # Find index of feature columns with correlation greater than 0.95
19  to_drop = [column for column in upper.columns if any(upper[column] > 0.95)]
20
21  # Drop features
22  X_train = X_train.drop(to_drop, axis = 1)
23  X_test = X_test.drop(to_drop, axis = 1)
```

06ModelDesign&Selection.py hosted with ❤ by GitHub view raw

Script 6 — Removing highly correlated features. To use a threshold of 0.90, change 0.95 to 0.90 in line 19.

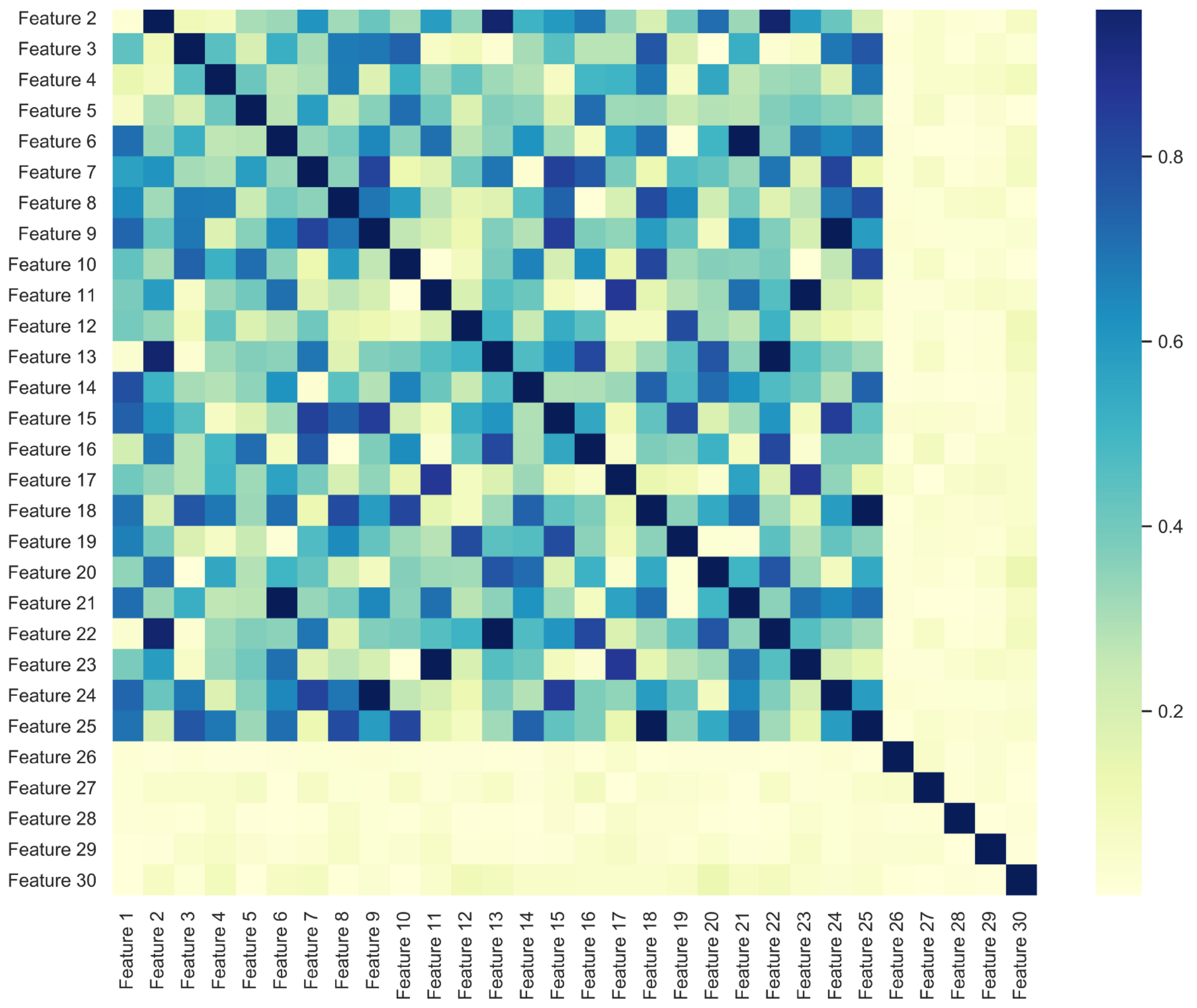


Figure 1 — Spearman's correlation matrix. Notice that the last 5 features don't correlate with any other feature since they are filled with random noise.

This should remove 6 features from the data set, which is not a lot — namely, features 13 and feature 21–25. However, in real data sets I've worked with, this step has reduced the number of features by up to 50 %. Just note that if you have thousands of features, this might be computationally expensive.

2. Wrapper Method: Recursive Feature Elimination with Cross Validation

After removing highly correlated features, we will further reduce the number of features by applying a recursive feature elimination algorithm. The Scikit-learn recursive feature elimination with cross validation (RFECV) object only allows you to use estimators/classifiers that have `feature_importances_` or `coef_` attributes. From experience, I've notice that RFECV often overestimates the number of features you really need.

First you need to select the base estimator to use with RFECV. For the sake of illustration, I will select a Random Forest Classifier as the base. You're welcome to choose any of the following as the base estimator.

```
###############################################################################
#                                Base Estimators                              #
###############################################################################
# Create dict for classifiers with feature_importances_ attribute
FEATURE_IMPORTANCE = {"Gradient Boosting", "Extra Trees Ensemble", "Random Forest"}
```

RFECV_baseEstimators.py hosted with ❤ by GitHub view raw

This is executed in the Classifiers section.

Once the base estimator is determined, we will tune its hyper-parameters. The reasons to do this are to reduce the risk over-fitting and to maximize the estimator's performance. To do so, we will create a Scikit-learn Pipeline object that will be used with the Scikit-learn GridSearchCV object.

GridSearchCV will perform an exhaustive search over the hyper-parameter grid and will report the hyper-parameters that will maximize the cross-validated classifier performance. Here is a nice Medium article showing a more detailed explanation. We will set the number of folds to 5.

The following are the steps in our Pipeline.

Step 1 — Feature Scaling: It's a common task to scale your features before using them in your algorithm. This is done to ensure that all the features in your data set have the same scale. Therefore, features with larger values won't dominate over features with smaller values. You can refer to this article for a more thorough explanation. We will use the samples in the training set to scale the data (training and test) *via* Z-score normalization. All features are centered around zero and have a standard deviation of 1.

Step 2 — Classifier: Defining the Classifier object to use in the Pipeline.

```
###############################################################################
#                    7. Tuning a classifier to use with RFECV                 #
###############################################################################
```

```
 7
 8  # Tune classifier (Took = 4.8 minutes)
 9
10  # Scale features via Z-score normalization
11  scaler = StandardScaler()
12
13  # Define steps in pipeline
14  steps = [("scaler", scaler), ("classifier", classifier)]
15
16  # Initialize Pipeline object
17  pipeline = Pipeline(steps = steps)
18
19  # Define parameter grid
20  param_grid = parameters[selected_classifier]
21
22  # Initialize GridSearch object
23  gscv = GridSearchCV(pipeline, param_grid, cv = 5,  n_jobs= -1, verbose = 1, scoring = '
24
25  # Fit gscv
26  print(f"Now tuning {selected_classifier}. Go grab a beer or something.")
27  gscv.fit(X_train, np.ravel(y_train))
28
29  # Get best parameters and score
30  best_params = gscv.best_params_
31  best_score = gscv.best_score_
32
33  # Update classifier parameters
34  tuned_params = {item[12:]: best_params[item] for item in best_params}
35  classifier.set_params(**tuned_params)
```

07ModelDesign&Selection.py hosted with ❤ by GitHub view raw

Script 7 — Tuning the classifier. To change the base estimator, change the classifier label in line 5. See the Classifiers section to see the list of available labels. To change the number of folds that GridSearchCV uses to 10, set cv = 10, in line 23. Similarly, you can also change the scoring.

The processing time to tune the Random Forest classifier took 4.8 minutes.

2.b. Recursively Selecting Features with the Tuned Estimator

Once we tuned our base estimator, we will create another pipeline similar to the first one, but this one will have the tuned classifier in the second step. Now a technicality arises. Since Scikit-learn Pipeline object does not have `feature_importances_` or `coef_` attributes, we will have to create our own pipeline object if we want to use it with

```
#                   8. Custom pipeline object to use with RFECV                #
################################################################################
# Select Features using RFECV
class PipelineRFE(Pipeline):
    # Source: https://ramhiser.com/post/2018-03-25-feature-selection-with-scikit-learn-
    def fit(self, X, y=None, **fit_params):
        super(PipelineRFE, self).fit(X, y, **fit_params)
        self.feature_importances_ = self.steps[-1][-1].feature_importances_
        return self
```

08ModelDesign&Selection.py hosted with ❤ by GitHub view raw

Script 8 — Defining a custom Pipeline object that is compatible with RFECV and RFE.

Finally, we can use RFECV with our new pipeline. Phew! Go grab a beer, mate.

```
################################################################################
#   9. Feature Selection: Recursive Feature Selection with Cross Validation   #
################################################################################
# Define pipeline for RFECV
steps = [("scaler", scaler), ("classifier", classifier)]
pipe = PipelineRFE(steps = steps)

# Initialize RFECV object
feature_selector = RFECV(pipe, cv = 5, step = 1, scoring = "roc_auc", verbose = 1)

# Fit RFECV
feature_selector.fit(X_train, np.ravel(y_train))

# Get selected features
feature_names = X_train.columns
selected_features = feature_names[feature_selector.support_].tolist()
```

09ModelDesign&Selection.py hosted with ❤ by GitHub view raw

Script 9 — Selecting features using a recursive feature elimination with cross-validation (RFECV).

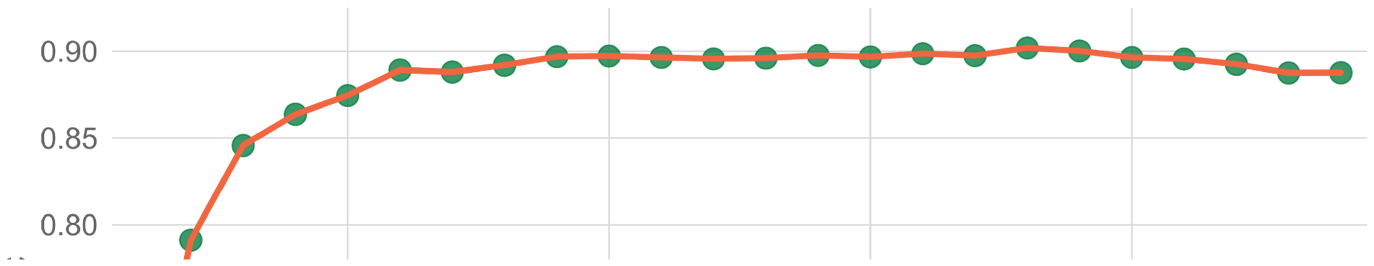
Now let's visualize the results. Plotting in python is kinda crazy, but whatever.

```
################################################################################
#                             10. Performance Curve                            #
################################################################################
# Get Performance Data
performance_curve = {"Number of Features": list(range(1, len(feature_names) + 1)),
```

```python
# Performance vs Number of Features
# Set graph style
sns.set(font_scale = 1.75)
sns.set_style({"axes.facecolor": "1.0", "axes.edgecolor": "0.85", "grid.color": "0.85",
               "grid.linestyle": "-", 'axes.labelcolor': '0.4', "xtick.color": "0.4",
               'ytick.color': '0.4'})
colors = sns.color_palette("RdYlGn", 20)
line_color = colors[3]
marker_colors = colors[-1]

# Plot
f, ax = plt.subplots(figsize=(13, 6.5))
sns.lineplot(x = "Number of Features", y = "AUC", data = performance_curve,
             color = line_color, lw = 4, ax = ax)
sns.regplot(x = performance_curve["Number of Features"], y = performance_curve["AUC"],
            color = marker_colors, fit_reg = False, scatter_kws = {"s": 200}, ax = ax)

# Axes limits
plt.xlim(0.5, len(feature_names)+0.5)
plt.ylim(0.60, 0.925)

# Generate a bolded horizontal line at y = 0
ax.axhline(y = 0.625, color = 'black', linewidth = 1.3, alpha = .7)

# Turn frame off
ax.set_frame_on(False)

# Tight layout
plt.tight_layout()

# Save Figure
plt.savefig("performance_curve.png", dpi = 1080)
```

10ModelDesign&Selection.py hosted with ❤ by GitHub view raw

Script 10 — Visualizing the results of RFECV. All this code to make a Figure 2, yuck! You might need to adjust the xlim(), ylim(), and ax.axhline() if you change the data set or base estimator.



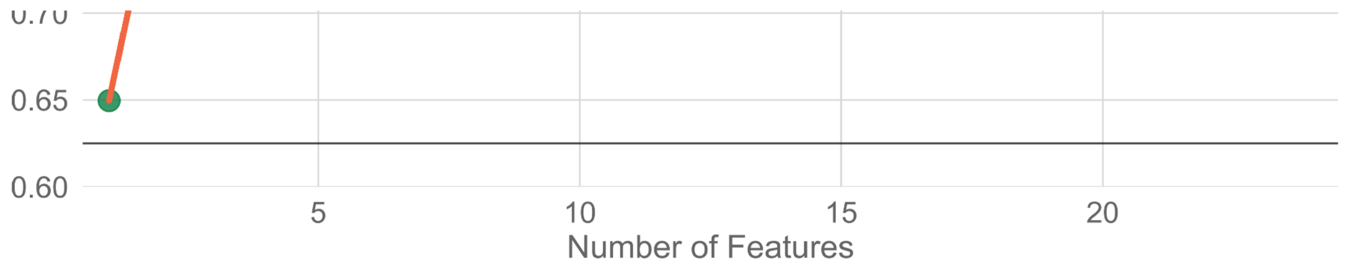


Figure 2 — Area under receiver operator curve (AUC) as a function of number of features. The classifier's performance peaks around 10 features.

What a beautiful figure! In **Figure 2**, we can see the classifier's performance as a function of a number of features. As you can see, the performance peak's around 10 features with an AUC of about 0.89; however, if you were to inspect the length of the selected_features list, you will notice that RFECV determined that you needed over 18 features to reach the peak performance.

It's problematic that we started with 30 features knowing that only 5 of them were truly necessary and that after our feature selection algorithm we ended up with over 18 representative features. To solve this problem, take a look at **Figure 2**, visually determine how many features you want to use (10 for example), and use the Scikit-learn RFE object with the `n_features_to_select` parameter set to 10. Notice that after 7 features the performance gain as features are added is minimal. You could use this as your threshold but I like to include a little redundancy since I do not know the optimal number of features for the other 17 classifiers. From Scikit-learn RFE documentation:

> *Given an external estimator that assigns weights to features (e.g., the coefficients of a linear model), the goal of recursive feature elimination (RFE) is to select features by recursively considering smaller and smaller sets of features … That procedure is recursively repeated on the pruned set until the desired number of features to select is eventually reached.*

```
###############################################################################
#                 11. Feature Selection: Recursive Feature Selection          #
###############################################################################
# Define pipeline for RFECV
steps = [("scaler", scaler), ("classifier", classifier)]
pipe = PipelineRFE(steps = steps)

# Initialize RFE object
```

Read more stories this month when you create a free Medium account.

```python
feature_selector.fit(X_train, np.ravel(y_train))

# Get selected features labels
feature_names = X_train.columns
selected_features = feature_names[feature_selector.support_].tolist()
```

11ModelDesign&Selection.py hosted with ❤ by GitHub view raw

Script 11 — Using recursive feature elimination (RFE) to select a given number of features. To change the size of the selected features to 12, set n_features_to_select =12 in line 9.

Now you might be wondering why didn't we use RFE to begin with instead of RFECV. Well, in real life scenarios, you will not know beforehand how many features you will really need. By using RFECV we are able to obtain the optimal subset of features; however, it's been my experience that it oftentimes overestimates. Nevertheless, from RFECV we obtain the performance curve from which we can make an informed decision of how many features we need. A disadvantage of using RFE is that the results are not cross-validated.

Feature Importance

Once we determine the selected features, we can investigate their importance according to the classifier. I speculate that some of the redundant features are actually more informative to the classifier than the real features. Let's see if that's true.

We will first train the tuned Random Forest classifier with the selected features. Then we will use the `feature_importances_` attribute and create a bar plot with it. **Note that the following code will only work if the classifier you selected as the base contains a `feature_importances_` attribute.**

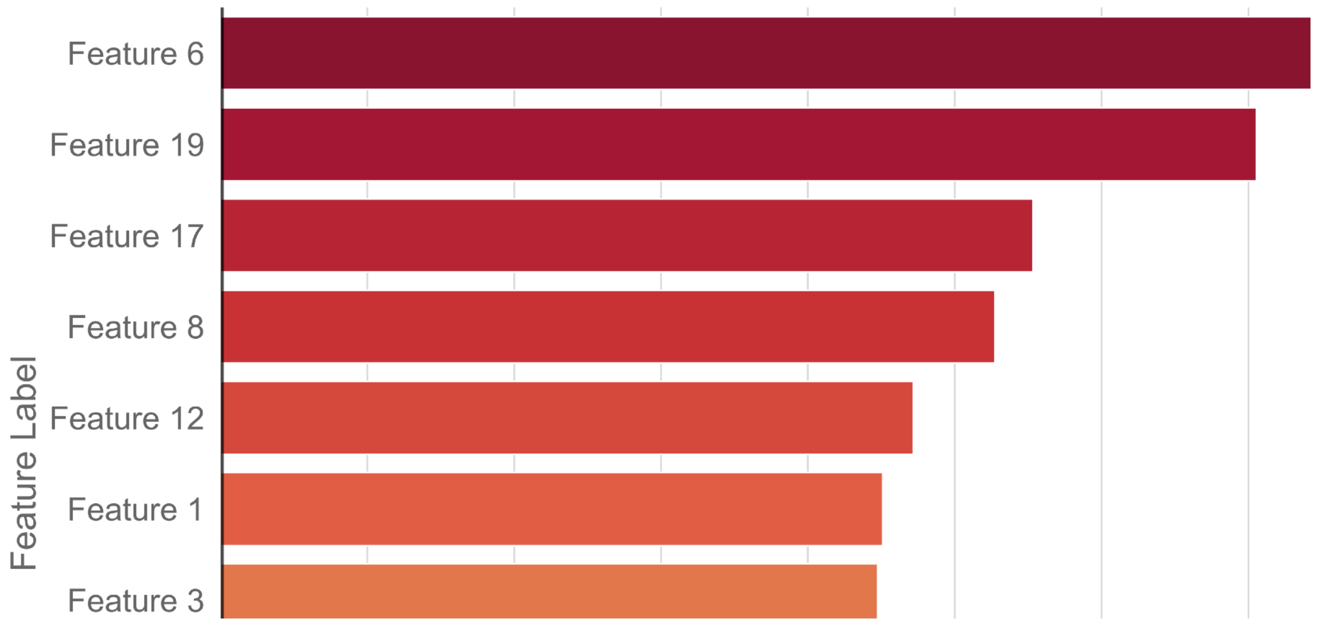
```python
###############################################################################
#                   12. Visualizing Selected Features Importance              #
###############################################################################
# Get selected features data set
X_train = X_train[selected_features]
X_test = X_test[selected_features]

# Train classifier
classifier.fit(X_train, np.ravel(y_train))

# Get feature importance
```

Read more stories this month when you create a free Medium account.

```
# Sort by feature importance
feature_importance = feature_importance.sort_values(by="Feature Importance", ascending=

# Set graph style
sns.set(font_scale = 1.75)
sns.set_style({"axes.facecolor": "1.0", "axes.edgecolor": "0.85", "grid.color": "0.85",
               "grid.linestyle": "-", 'axes.labelcolor': '0.4', "xtick.color": "0.4",
               'ytick.color': '0.4'})

# Set figure size and create barplot
f, ax = plt.subplots(figsize=(12, 9))
sns.barplot(x = "Feature Importance", y = "Feature Label",
            palette = reversed(sns.color_palette('YlOrRd', 15)),  data = feature_import

# Generate a bolded horizontal line at y = 0
ax.axvline(x = 0, color = 'black', linewidth = 4, alpha = .7)

# Turn frame off
ax.set_frame_on(False)

# Tight layout
plt.tight_layout()

# Save Figure
plt.savefig("feature_importance.png", dpi = 1080)
```

12ModelDesign&Selection.py hosted with ❤ by GitHub view raw

Script 12 — Visualizing the feature importances.



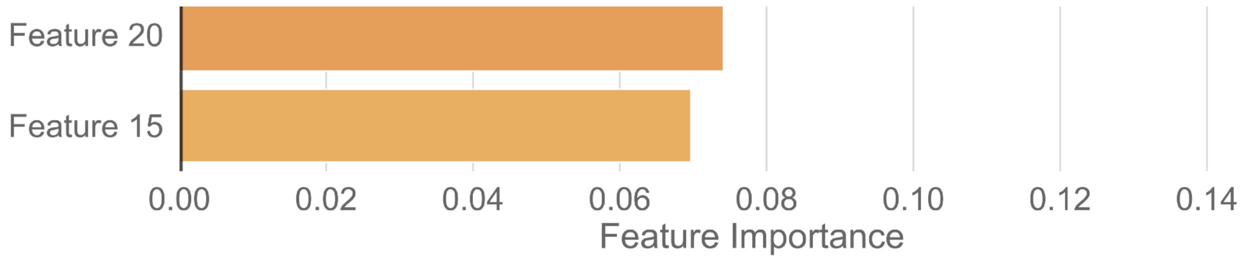


Figure 3 — Random Forest feature importance. Large values imply a greater importance.

Among the most important features are feature 6 and 19 which belong to the class of redundant features. It might seem counter-intuitive that the redundant features seem to be more important than the informative features (features 1–5). Then again, it can often be seen in Kaggle competitions that feature engineering can give you a boost. It's important to note that feature importances assigned by a machine learning classifier that is random in nature are not robust. For example, if you were to rerun RFE, you might obtain slightly different results since we didn't fix the seed in the Random Forest. That's why you need to cross-validate your results if your plan is to draw some conclusion from your feature importances. Here is an excellent article that talks about the randomness of machine learning. Additionally, do not trust the feature importances if your classifier is not tuned. Here is an amazing article about how to determine more robustly the feature importances.

Iterative Classifier Tuning and Evaluation

Now that we determined a subset of representative features, we are going to tune and train 18 models to investigate the highest performing models among them. To do this, we will iterate over the classifiers defined in **Script 4** and use **Script 7** to tune them using the hyper-parameters defined in **Script 5**. We will make minor changes to **Script 7** and add a few additional lines of code to evaluate the tuned classifier performance on a test set and save the results.

```
###############################################################################
#                      13. Classifier Tuning and Evaluation                  #
###############################################################################
# Initialize dictionary to store results
results = {}
```

Read more stories this month when you create a free Medium account.

```
    # Print message to user
    print(f"Now tuning {classifier_label}.")

    # Scale features via Z-score normalization
    scaler = StandardScaler()

    # Define steps in pipeline
    steps = [("scaler", scaler), ("classifier", classifier)]

    # Initialize Pipeline object
    pipeline = Pipeline(steps = steps)

    # Define parameter grid
    param_grid = parameters[classifier_label]

    # Initialize GridSearch object
    gscv = GridSearchCV(pipeline, param_grid, cv = 5,  n_jobs= -1, verbose = 1, scoring

    # Fit gscv
    gscv.fit(X_train, np.ravel(y_train))

    # Get best parameters and score
    best_params = gscv.best_params_
    best_score = gscv.best_score_

    # Update classifier parameters and define new pipeline with tuned classifier
    tuned_params = {item[12:]: best_params[item] for item in best_params}
    classifier.set_params(**tuned_params)

    # Make predictions
    if classifier_label in DECISION_FUNCTIONS:
        y_pred = gscv.decision_function(X_test)
    else:
        y_pred = gscv.predict_proba(X_test)[:,1]

    # Evaluate model
    auc = metrics.roc_auc_score(y_test, y_pred)

    # Save results
    result = {"Classifier": gscv,
              "Best Parameters": best_params,
              "Training AUC": best_score,
              "Test AUC": auc}

    results.update({classifier_label: result})
```

Script 13 took about 30 minutes to run in my workstation. I estimate that in a duo core CPU this would take about 3 hours. All the results will be stored in the dictionary object named `results`. The contents of the `results` dictionary can be accessed by the classifier_label (see **Classifiers** section). For each classifier, we store the following objects:

- **Classifier**: Pipeline object with trained classifier. You can use this to make predictions on new samples.
- **Best Parameters**: Dictionary containing the parameters that obtained the greatest performance in the training set.
- **Training AUC**: The cross-validated AUC obtained in the training set.
- **Test AUC**: The AUC obtained in the test set.

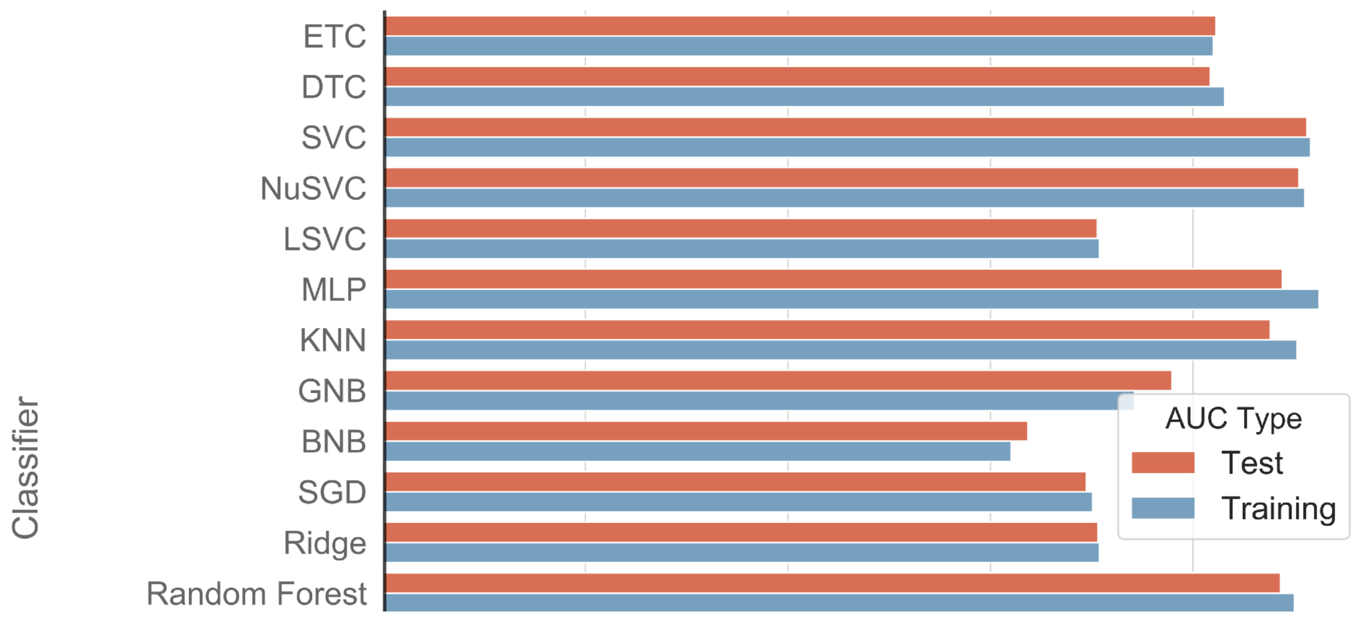
Lets visualize the results:

```
################################################################################
#                              14. Visualing Results                           #
################################################################################
# Initialize auc_score dictionary
auc_scores = {
              "Classifier": [],
              "AUC": [],
              "AUC Type": []
              }

# Get AUC scores into dictionary
for classifier_label in results:
    auc_scores.update({"Classifier": [classifier_label] + auc_scores["Classifier"],
                       "AUC": [results[classifier_label]["Training AUC"]] + auc_scores[
                       "AUC Type": ["Training"] + auc_scores["AUC Type"]})

    auc_scores.update({"Classifier": [classifier_label] + auc_scores["Classifier"],
                       "AUC": [results[classifier_label]["Test AUC"]] + auc_scores["AUC
                       "AUC Type": ["Test"] + auc_scores["AUC Type"]})

# Dictionary to PandasDataFrame
auc_scores = pd.DataFrame(auc_scores)

# Set graph style
```

```
                 'ytick.color': '0.4'})


# Colors
training_color = sns.color_palette("RdYlBu", 10)[1]
test_color = sns.color_palette("RdYlBu", 10)[-2]
colors = [training_color, test_color]

# Set figure size and create barplot
f, ax = plt.subplots(figsize=(12, 9))

sns.barplot(x="AUC", y="Classifier", hue="AUC Type", palette = colors,
            data=auc_scores)

# Generate a bolded horizontal line at y = 0
ax.axvline(x = 0, color = 'black', linewidth = 4, alpha = .7)

# Turn frame off
ax.set_frame_on(False)

# Tight layout
plt.tight_layout()

# Save Figure
plt.savefig("AUC Scores.png", dpi = 1080)
```

14ModelDesign&Selection.py hosted with ❤ by GitHub view raw

Script 14 — Visualizing the results of Script 13.



Read more stories this month when you create a free Medium account.

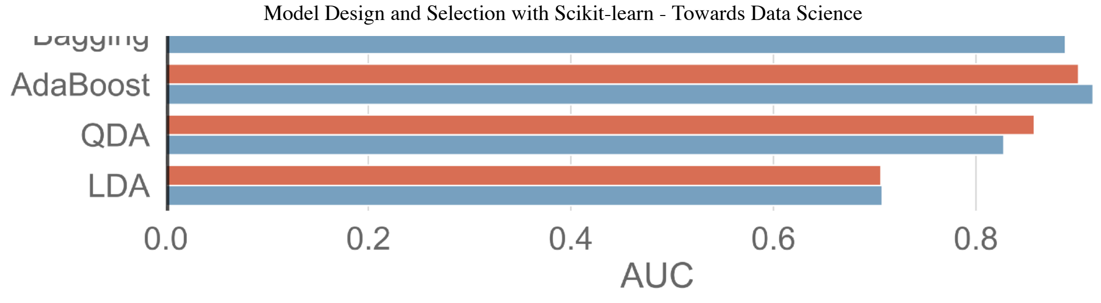


Figure 4 — Bar plot of classifier performance on training and test set.

From **Figure 4**, we can visually determine that SVC, NuSVC, Gradient Boosting, and AdaBoost classifiers obtained the highest performance in the test set. Look into the contents of the pandas dataframe object `auc_scores` to see the numerical results.

Closing Remarks

If you reached the end of this article, congrats! We actually went through a lot of material. I hope that this will be of aid to you. You can find all the code for this article in my GitHub repository. You're welcome to fork it. If you would like to use it, simply modify **Script 2** and make sure of the following:

- Load your data and encode all categorical variables.
- Take care of any missing values or outliers.
- Balance your data set (if needed).
- Store the feature matrix **X** into a pandas DataFrame object. Do the same for the targets in **y**.

If your data set contains about 1000 samples and 30 features, it should take about 30–45 minutes for the whole process to execute — assuming you have similar hardware than me.

Now some advice to determine what to do next to further improve the performance of these classifiers.

The easiest thing would be to select the top five performing classifiers and to run a Grid Search with different parameters. Once you have a feeling about where the best use them in VotingClassifier in Scikit-learn. This would most likely result in a higher performance but would increase the complexity of your modeling. You can also consider stacking — to learn more about it click here.

You can also look into feature engineering. This would give you the most bang for your buck. I'm planning on writing an article on this subject.

Find me at LinkedIn. Until next time!

Machine Learning Scikit Learn Python Data Science Modeling