Published in The Startup

J.P. Rinfret Follow

Dec 17, 2019 · 8 min read · ✦ · Listen

The Hyperparameter Cheat Sheet

A quick guide to hyperparameter tuning utilizing Scikit Learn's GridSearchCV, and the bias/variance trade-off

Before we dive in, let's start with a quick definition. In machine learning, a hyperparameter (sometimes called a tuning or training parameter) is defined as any parameter whose value is set/chosen at the onset of the learning process. Whereas other parameter values are computed during training.

In this blog, we will discuss some of the important hyperparameters involved in the following machine learning classifiers: K-Nearest Neighbors, Decision Trees and Random Forests, AdaBoost and Gradient Boost, and Sup[illegible]ines. Specifically, I will focus on the

hyperparameters that tend to have the greatest effect on the bias-variance tradeoff. Please note that the below lists are by no means meant to be exhaustive, and I encourage everyone to research each parameter on their own.

This blog assumes a basic understanding of each classifier, so we will skip a theory overview and mostly dive right into the tuning process utilizing Scikit Learn.

. . .

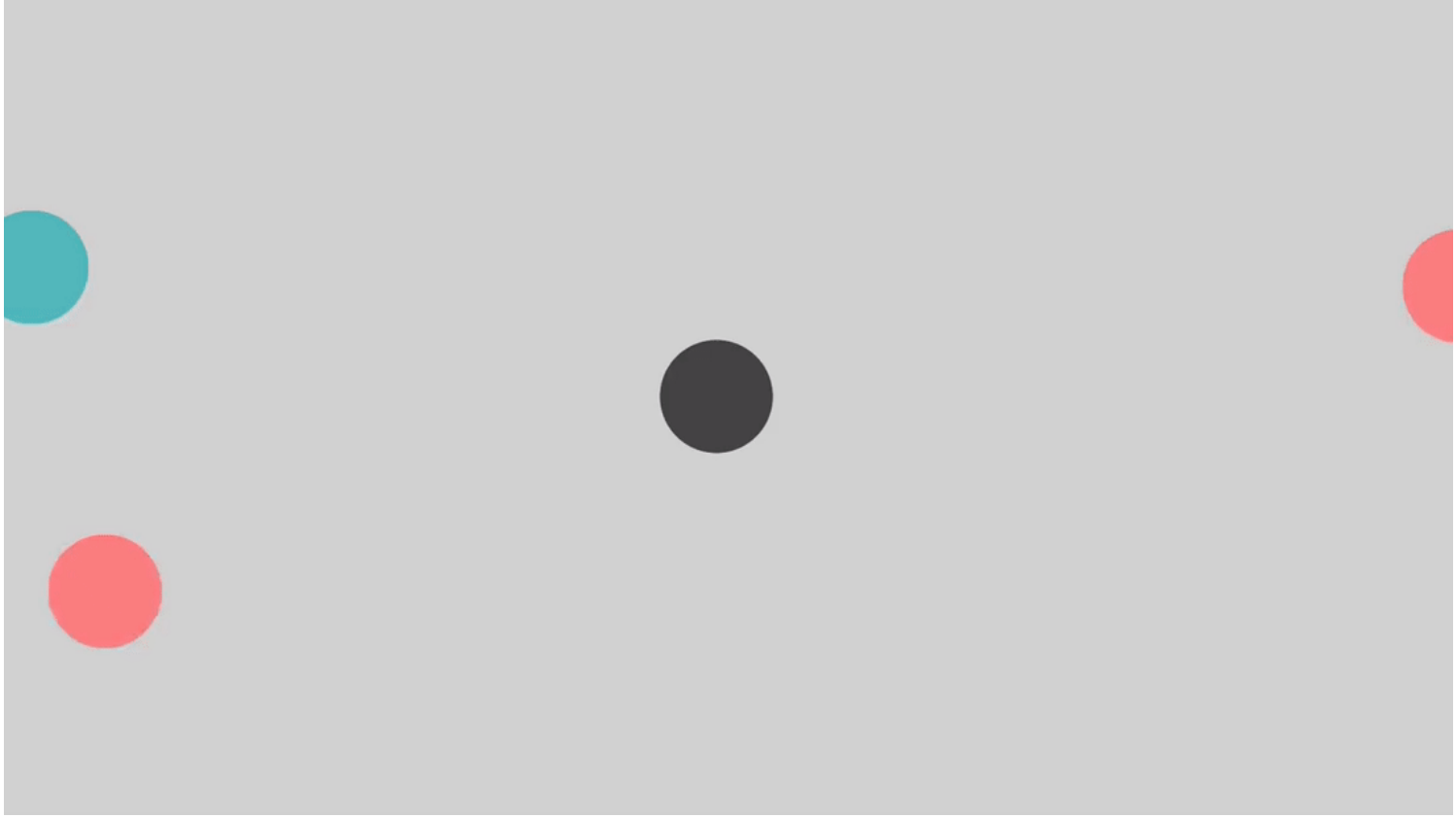
K-Nearest Neighbors (KNN)

Photo by Christian Stahl on Unsplash

In the KNN classifier (documentation), a data point is labeled based on its proximity to its neighbors. But how many neighbors should be considered in the classification?

N_neighbors (K)

Simply put, K is the number of neighbors that defines an unlabeled datapoint's classification boundary.



At K=3, the black datapoint is labeled as red since red has blue outnumbered 2:1

K takes in a range of integers (default = 5), finds the K-nearest neighbors, calculates the distance from each unlabeled point to those K-neighbors. How distance is calculated is defined by the metrics parameter explained below.

Important Note: K tends to be odd to avoid ties (i.e., if K = 4, this could result in a 2 Yes and 2 No, which would confuse the classifier).

Bias-Variance Tradeoff: in general, the smaller the K, the tighter the fit (of the model). Therefore, by decreasing K, you are decreasing bias and increasing variance, which leads to a more complex model.

Other Parameters for Consideration

- **Leaf_size** determines how many observations are captured in each leaf of either the BallTree of KDTree algorithms, which ultimately make the classification. The default equals 30. You can tune leaf_size by passing in a range of integers, like n_neighbors, to find the optimal leaf size. It is important to note that leaf_size can have a serious effect on run time and memory usage. Because of this, you tend not to run it on leaf_sizes smaller than 30 (smaller leafs equates to more leafs).

- **Weights** is the function that weights the data when making a prediction. “Uniform” is an equal weighted function, while “distance” weights the points by the inverse of their distance (i.e., location matters!). Utilizing the “distance” function will result in closer data points having a more significant influence on the classification.

- **Metric** can be set to various distance metrics (see here) like Manhattan, Euclidean, Minkowski, or weighted Minkowski (default is “minkowski” with a p=2, which is the Euclidean distance). Which metric you choose is heavily dependent on what question you are trying to answer.

- Note: we will skip the algorithm hyperparameter because it is preferred to set this parameter to “auto” so that the computer tells you which tree algorithm is best. I would recommend reading sklearn’s documentation for more information on the differences between the BallTree and KDTree algorithms.

Sample K optimizing code block (assumes F1 is best scoring metric

```
# find the optimal k
best_k = 0
best_score = 0
neighbors = range(1,10,2) # will consider min_k = 1, max_k = 25, only odd numbers
for k in neighbors:
    knn = KNeighborsClassifier(n_neighbors = k) # instantiate classifier
    knn.fit(X_train, y_train) # fit model
    knn_y_pred = knn.predict(X_test) # make a prediction

    # we will consider the optimal K to be the K that produces the highest f1 score
    f1 = metrics.f1_score(y_test, knn_y_pred)
    if f1 > best_score:
        best_k = k
        best_score = f1

# instantiate the classifier with the optimal K, fit model, and make prediction
knn = KNeighborsClassifier(n_neighbors = best_k)
knn.fit(X_train, y_train)
best_knn_pred = knn.predict(X_test)
```

Optimizing for K

. . .

Decision Trees and Random Forests

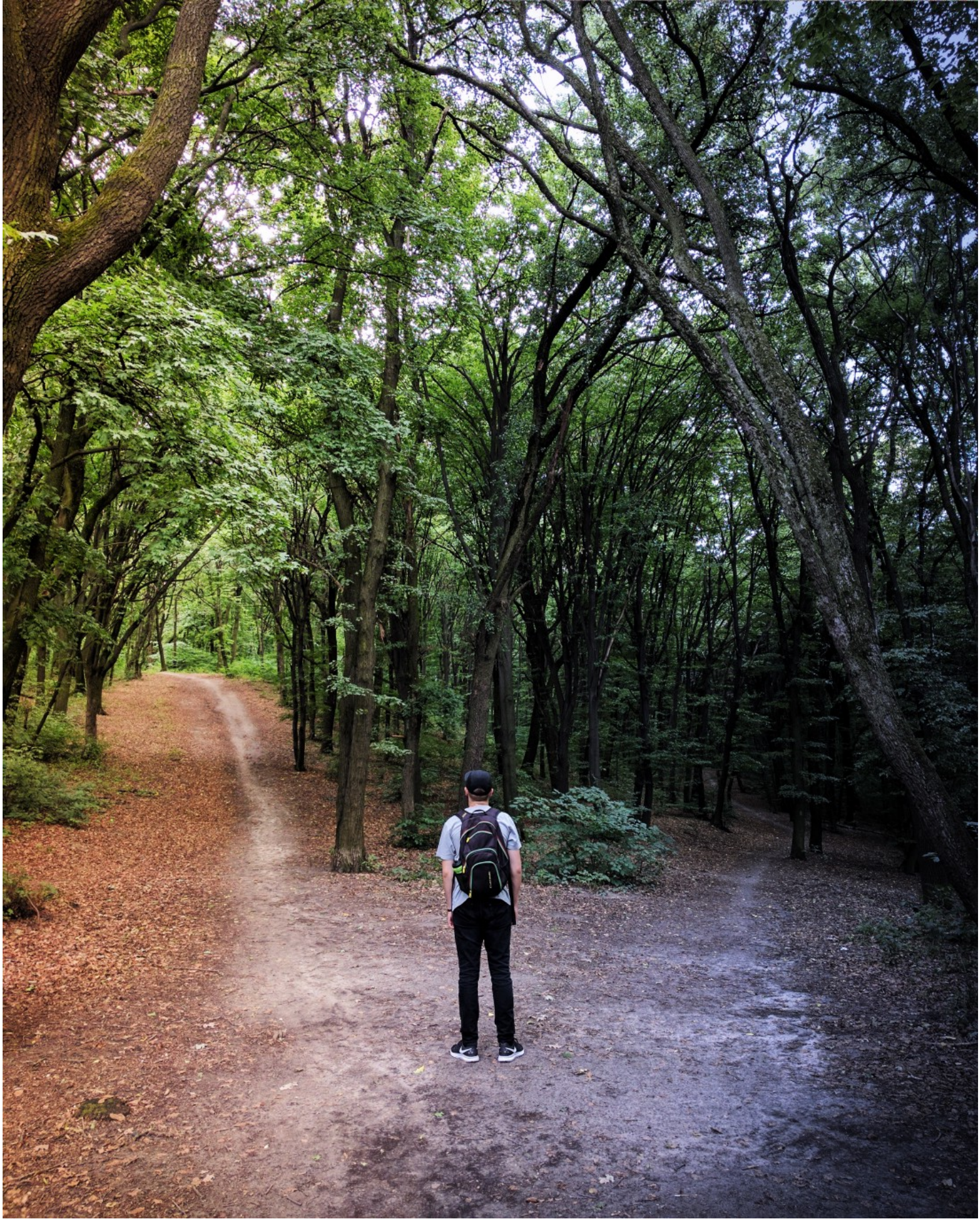


Photo by Vladislav Babienko on Unsplash

When building a Decision Tree (documentation) and/or Random Forest (documentation), there are many important hyperparameters to be considered. Some of these are found in both classifiers while some are just specific to forests.

Let's take a look at the hyperparameters that are most likely to have the largest effect on bias and variance.

- **N_estimators** (only used in Random Forests) is the number of decision trees used in making the forest (default = 100). Generally speaking, the more uncorrelated trees in our forest, the closer their individual errors get to averaging out. However, more does not mean better since this can have an exponential effect on computation costs. After a certain point, there exists statistical evidence of diminishing returns. **Bias-Variance Tradeoff:** in theory, the more trees, the more overfit the model (low bias). However, when coupled with bagging, we need not worry.

- **Max_depth** is an integer that sets the maximum depth of the tree. The default is None, which means the nodes are expanded until all the leaves are pure (i.e., all the data belongs to a single class) or until all leaves contain less than the min_samples_split, which we will define next. **Bias-Variance Tradeoff:** increasing the max_depth leads to overfitting (low bias)

- **Min_samples_split** is the minimum number of samples required to split an internal node. **Bias-Variance Tradeoff:** the higher the minimum, the more "clustered" the decision will be, which could lead to underfitting (high bias).

- **Min_samples_leaf** defines the minimum number of samples needed at each leaf. The default input here is 1. **Bias-Variance Tradeoff:** similar to min_samples_split, if you do not allow the model to split (say because your min_samples_lear parameter is set too high) your model could be over generalizing the training data (high bias).

Other Parameters for Consideration

- **Criterion** measures the quality of the split and receives either "gini", for Gini impurity (default), or "entropy", for information gain. Gini impurity is the probability of *incorrectly* classifying a randomly chosen datapoint if it were labeled according to the class distribution of the dataset. Entropy is a measure of chaos in your data set. If a split in the dataset results in lower entropy, then you have gained information (i.e., your data has become more decision useful) and the split is worthy of the additional computational costs.

GridSearchCV Optimization for Random Forests

```python
# imports
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV

# define the hyperparameters we want to tune
param_grid = {
    'criterion': ['gini', 'entropy'],
    'n_estimators': [100, 150, 200],
    'max_depth': [None, 1, 3, 5, 10],
    'min_samples_split': [5, 10],
    'min_samples_leaf': [5, 10]
}

# instantiate GridSearchCV, fit model, and make prediction
gs_rf = GridSearchCV(RandomForestClassifier(), param_grid = param_grid)
gs_rf.fit(X_train, y_train)
y_pred = gs_rf.predict(X_test)
```

A sample block of code illustrating how to tune the Random Forest Classifier

. . .

AdaBoost and Gradient Boosting

Photo by Michelle Bonkosky on Unsplash

By utilizing weak learners (aka “stumps”), boosting algos like AdaBoost (documentation) and Gradient Boosting (documentation) focus on what the model misclassifies. By overweighting these misclassified data points, the model focuses on what it got wrong in order to learn how to get them right.

Similar to Decision Trees and Random Forests, we will focus on the bias-variance tradeoff usual suspects.

- **N_estimators** is the maximum number of estimators at which boosting is terminated. If a perfect fit is reached, the algo is stopped. The default here is 50. **Bias-Variance Tradeoff:** the higher the number of estimators in your model the lower the bias.

Other Important Parameters

- **Learning_rate** is the rate at which we are adjusting the weights of our model with respect to the loss gradient. In layman’s terms: the lower the learning_rate, the slower we travel along the slope of the loss function. **Important note:** there is a trade-off between learning_rate and n_estimators as a tiny learning_rate and a large n_estimators will not necessarily improve results relative to the large computational costs.
- **Base_estimator (AdaBoost) / Loss (Gradient Boosting)** is the base estimator from which the boosted ensemble is built. For AdaBoost the default value is None, which equates to a Decision Tree Classifier with max depth of 1 (a stump). For Gradient Boosting the default value is deviance, which equates to Logistic Regression. If “exponential” is passed, the AdaBoost algorithm is used.

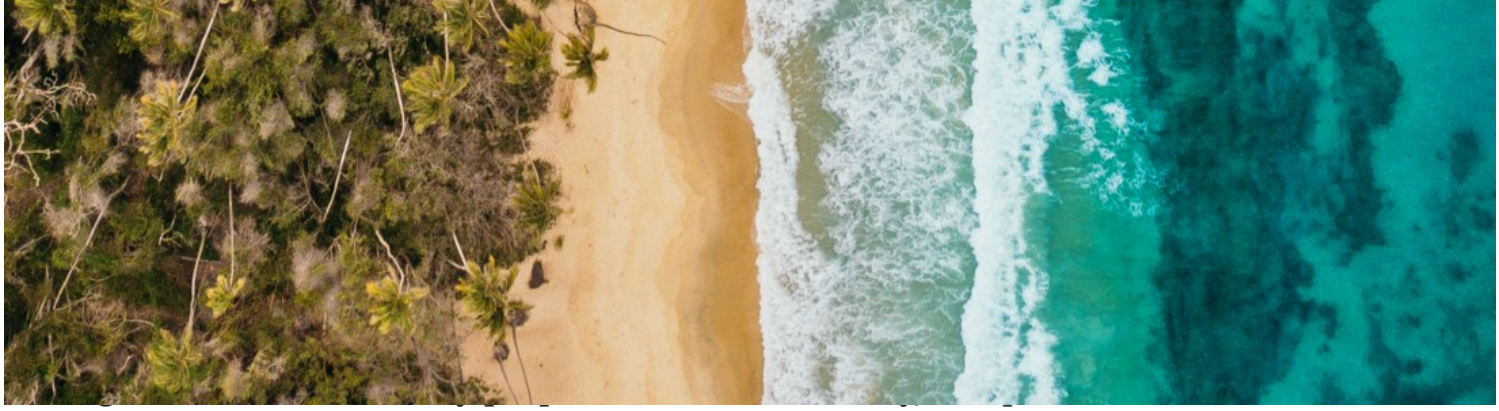
GridSearchCV Optimization for AdaBoost

```
# imports
from sklearn.ensemble import AdaBoostClassifier
from sklearn.model_selection import GridSearchCV

# define the hyperparameters we want to tune
param_grid = {
    'n_estimators': [100, 200],
    'learning_rate' : [0.001, 0.01, 0.1, 0.2, 0.5]
}

# instantiate GridSearchCV, fit model, and make prediction
gs_ab = GridSearchCV(AdaBoostClassifier(), param_grid = param_grid)
gs_ab.fit(X_train, y_train)
y_pred = gs_ab.predict(X_test)
```

A sample block of code illustrating how to tune the AdaBoost Classifier

. . .

Support Vector Machines (SVM)



much you want to avoid being wrong. You can think of the inverse of C as your total error budget (summed across all training points), with a lower C value allowing for more error than a higher value of C. **Bias-Variance Tradeoff:** as previously mentioned, a lower C value allows for more error, which translates to higher bias.

- **Gamma** determines how far the scope of influence of a single training points reaches. A low gamma value allows for points far away from the hyperplane to be considered in its calculation, whereas a high gamma value prioritizes proximity. **Bias-Variance Tradeoff:** think of gamma as inversely related to K in KNN, the higher the gamma, the tighter the fit (low bias).

A Very Important Hyperparameter

- **Kernel** specifies which kernel should be used. Some of the acceptable strings are “linear”, “poly”, and “rbf”. Linear uses linear algebra to solve for the hyperplane, while poly uses a polynomial to solve for the hyperplane in a higher dimension (see Kernel Trick). RBF, or the radial basis function kernel, uses the distance between the input and some fixed point (either the origin or some of fixed point c) to make a classification assumption. More information on the Radial Basis Function can be found here.

GridSearchCV Optimization for SVM.

```
# imports
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV

# define the hyperparameters we want to tune
param_grid = {
    'kernel' : ['linear', 'poly', 'rbf'],
    'C' : [0.001, 0.01, 0.1, 1, 10],
    'gamma' : [0.001, 0.01, 0.1, 1],
}

# instantiate GridSearchCV, fit model, and make prediction
gs_svc = GridSearchCV(SVC(), param_grid = param_grid)
gs_svc.fit(X_train, y_train)
y_pred = gs_svc.predict(X_test)
```

A sample block of code illustrating how to tune the Support Vector Classifier

Final Word

We have seen multiple was to train a model using sklearn, specifically GridSearchCV. However, it is very, very important to keep in mind the bias-variance tradeoff, as well as the tradeoff between computational costs and scoring metrics. Ideally, we want a model with low bias and low variance to limit overall error, but is it really worth the extra run-time, memory, etc. for only slight improvements? I will let you answer that.

Machine Learning | Data Science | Hyperparameter Tuning | Data Engineering | Statistical Learning