Water Quality and Potability Analysis

DSP 556 — Nellie Dawson, Ryan Soucy

ABSTRACT:

As we recognize the current need and future demand for effective water infrastructures, we explored the impact of some select water quality variables on water potability. Through building 4 base classification models and 4 ensemble classification models, we set out to determine how accurately we are able to predict water potability. Through this analysis, we found some challenges in both data quality and classification accuracy.

BACKGROUND:

Community access to potable water has become increasingly topical over the past decade. Since the 2014 Flint water crisis in Michigan, a number of communities with known drinking water issues have also been acknowledged. In addition to the pollutants and contaminants already in these community water systems, climate change has both direct and indirect effects on potable water. Higher temperatures associated with climate change will naturally raise the demand for drinking water thus increasing the demand for effective infrastructure and community tools. In particular, higher temperatures in water decrease the pH level, meaning more acidity, which has been seen to increase "lead mobility and bioavailability" leading to the potential of more human and animal consumption of these toxic chemicals (Levin, et al., 2023). It is crucial that we understand the impact that water quality has on water potability.

As machine learning develops, there is more opportunity to use these models in many fields, including the processes of water quality monitoring and prediction. One study, *A review of the application of machine learning in water quality evaluation,* assesses the performance of 45 machine learning algorithms in a variety of different water sources – surface water, groundwater, drinking water, wastewater, and marine environments. The challenges they found were in accessibility to high-quality data, the generalizability of the models to unique infrastructures, and the accessibility of these models and tools overall. (Zhu, et al., 2022)

Another report, *A Machine Learning-Based Water Potability Prediction Model by Using Synthetic Minority Oversampling Technique and Explainable AI*, uses the same dataset we do in an attempt to fit Support Vector Machines, Decision Trees, Random Forest, Gradient Boost, and Ada Boost Classification methods. (Hindawi, (Patel, et al., 2022) These data scientists saw between 70 - 80% accuracy on their classification methods after extensive hyperparameter tuning. Although we have chosen some of the same models to build, this group implemented more hyperparameter tuning methods, and we have included some additional ensemble models to our analysis.

DATA DESCRIPTION:

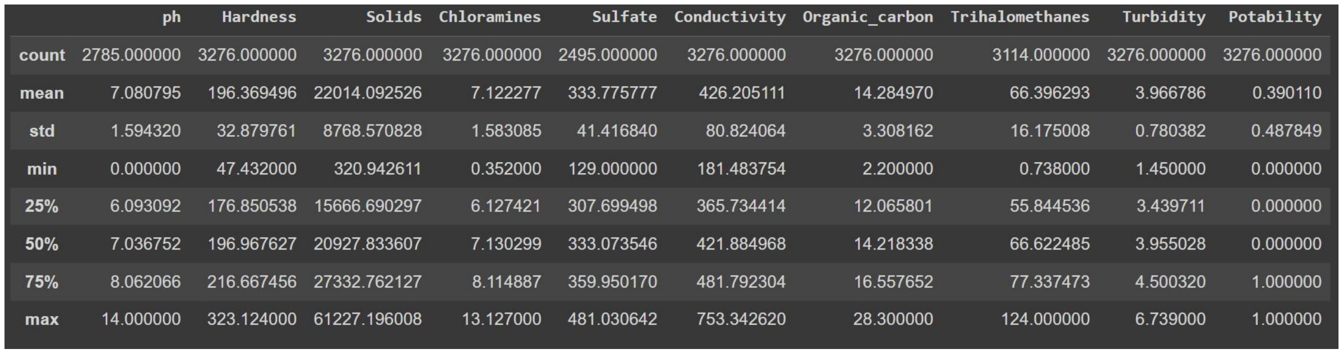
The water quality dataset we have selected is from Kaggle, and although this dataset has been distributed a handful of times on both Kaggle and GitHub, the original dataset was last updated by Aditya Kadiwal 3 years ago. The dataset contains 3,276 water sample observations on ten total variables — pH, Hardness, Solids, Chloramines, Sulfate, Conductivity, Organic carbon, Trihalomethanes, Turbidity, and Potability. The first nine of these variables are numeric values with units of measurement of mg/L (milligram per liter), μS/cm (microgram per liter), or ppm (parts per million). The final variable, potability, is a binary variable that indicates whether the water is potable or not with a 0 or 1. Each of the variables has a normal distribution for both the whole dataset and the potability subgroups, and have little to no collinearity. These variables have such normal distributions because the dataset is synthetically generated.

In order to determine if the data is still useful in learning from, we need to first explore the validity of this data before applying it to any models. Two issues that arose as we explored the data's validity were whether these values are distributed around expected ranges of these naturally occurring contaminants in water sources, and if the potability binary variable is just a function of the other features in this dataset. To address both of these issues, we explored the Environmental Protection Agency (EPA) National Primary Drinking Water Regulations and the World Health Organization (WHO) Guidelines for drinking water quality. These regulations and guidelines provide extensive documentation on each level of potential contaminant in water systems, various level recommendations for the most toxic contaminants, and recommendations for issues within water systems. The dataset used in this analysis accesses a subset of these potential contaminants and water quality parameters.

It is clear through our research that potability is not just a function of these particular values. From what hs been gathered, it seems as though most of these water quality measurements are not the most toxic contaminants in water systems. While some of these measurements do have EPA and WHO recommendations associated with them, many of these particular water quality measurements are said to have taste thresholds, and acceptability levels may vary in degree between communities.

To ensure each variable's distribution was around naturally occurring levels, we compared the distributions of each variable with their expected and/or recommended levels. The majority of variables in this dataset do hold distributions that match naturally occurring levels. On a scale of 0 to 14, **pH** indicates the level of acidity or alkalinity in the water, and most tap water has a pH level in the range of 6.5 to 8.5. According to the

World Health Organization, pH may not have a direct impact on consumers, but “it is one of the most important operational water quality parameters” (WHO, 2022). **Hardness** is a mg/L measurement of the amount of calcium and magnesium dissolved in the water. Both water that is too hard or too soft can have implications on the pipe systems. According to WHO Guidelines, the taste threshold for the hardness can be between 100–300 mg/L, but as acceptability varies for many taste thresholds in communities, in some cases, is consumed at levels greater than 500 mg/L. **Chloramines** are caused by the use of chlorine in water systems, and are recommended to not be above 4 ppm according to EPA regulations, and the taste thresholds of sodium **sulfate** are said to be approximately 250 mg/L according to WHO guidelines. The electrical **conductivity** of tap water has been seen in the range of 50 to 800 (μg/L) based on information from Atlas Standard. **Trihalomethanes** (THM) guideline levels are based on a ratio of a set of chemicals present when chlorine is in use in the filtration system, and levels up to 80 ppm are considered safe in drinking water. **Turbidity** is a measure of the cloudiness of the water, and the EPA says that depending on the filtration systems, turbidity should not be above 5 Nephelometric Turbidity Unit (NTU). As comparable to Figure 1, the levels of each of the above features in our dataset are in this range or within reasonable distributions. Because approximately 60% of our data is not potable, it is reasonable to see a means a bit higher than these taste thresholds in this synthetically created dataset.



	ph	Hardness	Solids	Chloramines	Sulfate	Conductivity	Organic_carbon	Trihalomethanes	Turbidity	Potability
count	2785.000000	3276.000000	3276.000000	3276.000000	2495.000000	3276.000000	3276.000000	3114.000000	3276.000000	3276.000000
mean	7.080795	196.369496	22014.092526	7.122277	333.775777	426.205111	14.284970	66.396293	3.966786	0.390110
std	1.594320	32.879761	8768.570828	1.583085	41.416840	80.824064	3.308162	16.175008	0.780382	0.487849
min	0.000000	47.432000	320.942611	0.352000	129.000000	181.483754	2.200000	0.738000	1.450000	0.000000
25%	6.093092	176.850538	15666.690297	6.127421	307.699498	365.734414	12.065801	55.844536	3.439711	0.000000
50%	7.036752	196.967627	20927.833607	7.130299	333.073546	421.884968	14.218338	66.622485	3.955028	0.000000
75%	8.062066	216.667456	27332.762127	8.114887	359.950170	481.792304	16.557652	77.337473	4.500320	1.000000
max	14.000000	323.124000	61227.196008	13.127000	481.030642	753.342620	28.300000	124.000000	6.739000	1.000000

Table 1

Through our research into water potability measures and the validity of this data, we determined that two variables in particular did not seem to accurately reflect levels found naturally in water systems: **Solids** and **Organic carbon**. According to the WHO Guidelines, the Total Dissolved Solids (TDS) should be around 600 - 1000 in potable water systems. This dataset has Solids quantities (ppm) that are clearly much higher than this expected value, with a mean of approximately 22,014. This value seems beyond a reasonable distribution for us to continue to include it in our analysis. Similarly, the Organic Carbon levels should be below 2 for safe water consumption. This dataset has values ranging from a minimum of 2.2 to a maximum of 28.3. Again, these values seem unrealistic for water quality samples and we use the panda's drop method to remove these variables before continuing with our analysis. Removing these two variables will not negatively affect our analysis because since this dataset is synthetically generated, each variable is normally distributed, there is no correlation between variables, and potability is not a function of these exact features.

When looking at the dataset further, some observations have null values on select variables. Most models cannot handle null values, so we had to make a decision about what to do with the cells in our data that are empty. After removing all observations that have null values on any variable, we were able to retain 2011 of the 3216 original water samples. Once we created 80/20 training and test splits, the data was scaled using the training data and fit to the test data to avoid data leakage. The data is also individually scaled using a pipeline for each GridSearch to avoid data leakage on models that will be affected by the variance of the different predictors, including Support Vector Machines, Logistic Regression, and Stacking.

METHODS:

Aside from the preprocessing abilities of pandas, we almost entirely utilize tools within sklearn to complete our analysis. With the goal of classification to a binary water potability variable, we built a variety of basic classification models as well as a handful of ensemble models. By doing so, we are able to compare model accuracy across various model types as well as across ensemble models that use Decision Trees as the default classifier. The primary focus of our analysis will be to tune models using cross-validated grid search in order to find the best model. We will compare DecisionTreeClassifier, RandomForestClassifier, SupportVectorClassifier, and LogisticRegression, then we will use Bagging and Boosting in conjunction with Decision Tree classification and compare these models to the success of the base estimators.

Basic Classification Models

For **Decision Trees,** we chose to adjust the parameters max_depth, min_samples_split, and ccp_alpha. The variable max_depth dictates the maximum depth of the tree. Normally a tree will continue until all leaves contain less than min_samples_split samples, but if we adjust the max_depth to be smaller, we are attempting to avoid overfitting to the noise at the smaller, more difficult to distinguish nodes. The variable min_samples_split dictates the minimum number of samples required to split an internal node. This determines how much power individual points may have over the node splitting especially as we approach the later nodes in the tree. The variable ccp_alpha is a cost-complexity parameter that prunes the tree to avoid over-fitting. For the **Random**

Forest model, the parameters we chose to adjust were max_depth, min_samples_split, and n_estimators. We will not adjust ccp_alpha for Random Forest as we found it not to be as powerful of a parameter of the others when exploring Decision Trees, but it gives us the opportunity to adjust the parameter that Random Forest introduces called the number of estimators which determines how many decision trees are in the Random Forest Model. For both the **SVC** model and the **Logistic Regression** model, we choose to adjust C, a regularization parameter that controls the balance between the margin size and training errors.

Ensemble Classification Models

After exploring these basic models, we shifted to ensemble models of bagging and boosting in an attempt to make some stronger models. We used sklearn models of BaggingClassifier, AdaBoostClassifier, GradientBoostingClassifier, and StackingClassifier. The **Bagging Classifier** uses DecisionTreeClassifier as a base estimator then trains each tree on a random redistribution of the training set. We have set the max_features to 7, the maximum number of numerical predictors/parameters in the dataset. The parameters we tuned are max_samples which dictates the maximum number of samples from X to train the base decision trees, and the variable n_estimators which determines the number of decision trees ran under the bagging classifier model. The **AdaBoost Classifier** also uses a base estimator of decision trees, and the parameters we tune for this model are learning_rate and n_estimators. Learning rate is the weight applied to each classifier at each boosting iteration, with higher values increasing the contribution of each classifier. The n_estimators is again the number of decision trees run in the model. For **Gradient Boosting**, we are using the familiar variables of learning_rate, n_estimators, max_depth. The **Stacking Classifier** uses a stack of multiple base estimators to attempt to reduce the bias from each of the individual models. The outputs from the first model are used as the input for the second, and so on. We selected base estimators of Decision Tree and Logistic Regression tuned to the best parameters from the base models. We use a final estimator of Support Vector Classification.

RESULTS:

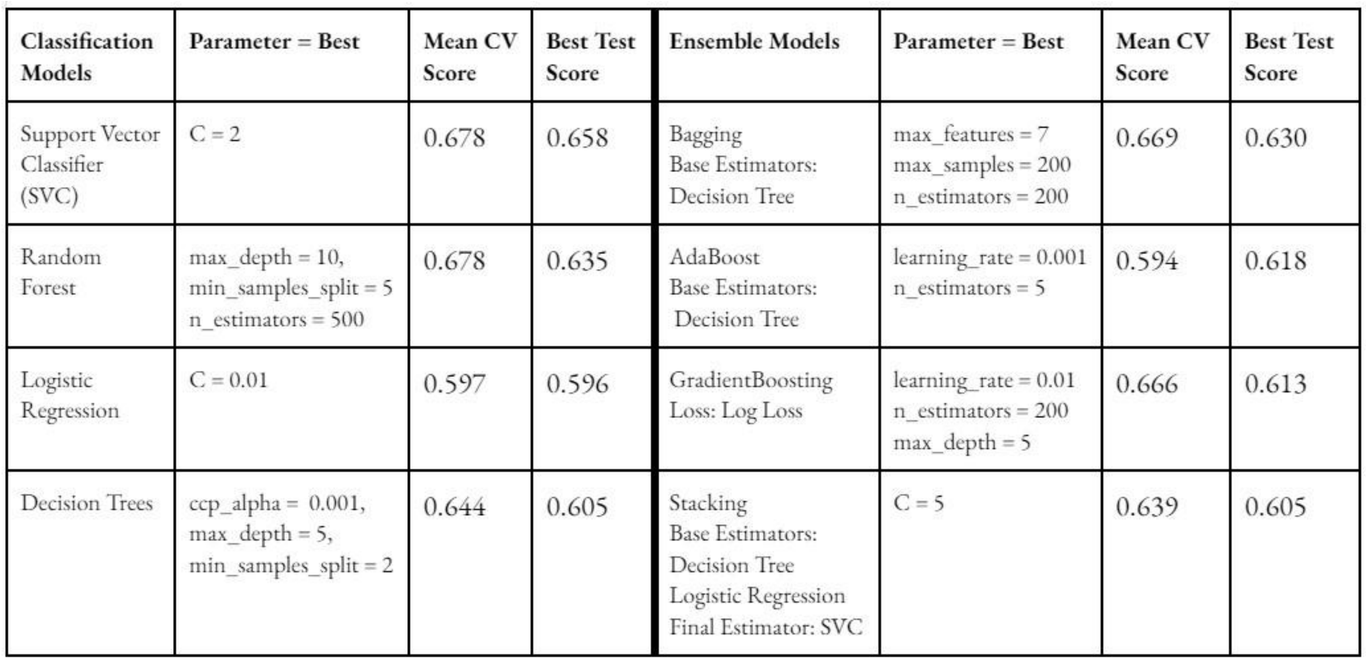
Basic Classification Models

After parameter tuning the **Decision Tree,** we see a test accuracy of 0.605 with the best parameters of ccp_alpha = 0.001, max_depth = 5, and min_samples_split = 2. We first saw a best max_depth of 12, but noticed we actually see a drop in accuracy as max_depth increases, and a comparison of train and test accuracy scores indicated we may be overfitting. Removing these larger depth values from our tuning saw similar training set results and improved test set results. Results from **Random Forest** were an improvement over Decision

Trees with a test accuracy of 0.635, with the best parameters of n_estimators = 500, max_depth = 10, and min_samples_split = 5. The max_depth of each tree within the random forest seems to have a significant impact on the accuracy, but again we see an accuracy decrease as max depth rises. Although these scores are not exceptional, the most successful model was the **Support Vector Classifier (SVC)** with a test accuracy of 0.658 using just the one regularization parameter. C does not have drastic effects on time or accuracy for SVC but controls the balance between the margin size and training errors. This C=2 indicates that the model is attempting to focus more on misclassification than on margin maximization. **Logistic Regression** saw an inversely sized best regularization parameter of 0.01, but it was the most poorly performing model we built in this analysis with a test set of just 0.596.

Ensemble Classification Models

After exploring these basic models, we then moved on to ensemble models of bagging and boosting to potentially obtain stronger models. We again used sklearn models of BaggingClassifier, AdaBoostClassifier, GradientBoostingClassifier, and StackingClassifier. While tuning the **Bagging Classifier,** we found that with max_feature = 7, and best parameters of max_samples = 200, and n_estimator = 200, we see a test score of 0.630. Although these scores are not exceptional, the model performs slightly better than the base estimator Decision Tree Classifier, which had a score of 0.605 and is comparable to Random Forest which had a test score of 0.635. By Bagging the Decision Tree Classifier, we hoped to test the data on a larger randomized population to determine the test accuracy. This showed us that our model accuracy is still quite low, but is avoiding overfitting and underfitting because the training and test scores are similar in magnitude. With the **AdaBoost Classifier** again using DecisionTrees as the base estimator, we found that with the best parameters of learning_rate = .001 and n_estimators = 5, we see a test score of 0.618. This model performs about as well on the test set as the Bagging model. We are a bit concerned that this model is potentially underfitting the data or, more likely, not running enough decision trees to decrease bias and variance due to the n_estimators only being 5. This is the optimal parameter, but it just does not seem to be maximizing the potential of the AdaBoost Classifier, so it feels worth noting. We found that with the best parameters of learning_rate = .01, max_depth = 5, and n_estimators = 200, the **GradientBoostClassifier** sees a test score of 0.613. This model performs about as well on the test set as the Bagging model and AdaBoosting model. The fit times of these models were all pretty low, but the max_depth did seem to have the most impact of a few seconds, which is pretty minimal. These tuned parameters also seem to be more reasonable than AdaBoosting's. Having relatively shallow trees with a max_depth of 5 may be underfitting the data on each individual tree, but it saves time in the model and the n_estimators of 200 decreases the impact of this underfitting by generating 200 trees within the model. This is exactly how the model is supposed to run. For the **StackingClassifier**, we tuned the final estimator of SVC to C=5. The base estimators of DecisionTreeClassifier and LogisticRegression were tuned using the best tuning parameters determined in the Base Models section of our report. The training cross-validated score of the model came out to .639, while the test accuracy score came out to .605. This is the lowest test score of all of the ensemble models, but it is not a significant difference.



Classification Models	Parameter = Best	Mean CV Score	Best Test Score	Ensemble Models	Parameter = Best	Mean CV Score	Best Test Score
Support Vector Classifier (SVC)	C = 2	0.678	0.658	Bagging Base Estimators: Decision Tree	max_features = 7 max_samples = 200 n_estimators = 200	0.669	0.630
Random Forest	max_depth = 10, min_samples_split = 5 n_estimators = 500	0.678	0.635	AdaBoost Base Estimators: Decision Tree	learning_rate = 0.001 n_estimators = 5	0.594	0.618
Logistic Regression	C = 0.01	0.597	0.596	GradientBoosting Loss: Log Loss	learning_rate = 0.01 n_estimators = 200 max_depth = 5	0.666	0.613
Decision Trees	ccp_alpha = 0.001, max_depth = 5, min_samples_split = 2	0.644	0.605	Stacking Base Estimators: Decision Tree Logistic Regression Final Estimator: SVC	C = 5	0.639	0.605

Table 2

DISCUSSION:

After tuning the hyperparameters of our various models using cross-validated grid searches, we were unable to create a highly accurate model to determine whether or not water was potable based on our predictor variables. We were able to create models that had some levels of accuracy, but our most accurate result was only .658 using the Support Vector Classifier. All of our models seemed to have approximately similar test set results, which we found quite interesting. No one model type seemed to be drastically better or worse than the others. Our lowest test set score was .596 using Logistic Regression, which is only .062 lower than our most accurate SVC model.

We did not see an increase in model accuracy through the use of ensemble models. The BaggingClassifier had the highest test score accuracy of the ensemble models at .630, which is lower than the SVC test score. We were not able to build more accurate models than the previous researchers mentioned that used this dataset which saw 70 - 80% accuracy on their classification methods, but they incorporated stronger hyperparameter tuning techniques and did not remove any variables as we did.

There are several avenues for future exploration from the research that we performed. Firstly, the methods that we used were all correct, despite low accuracy within the model. We believe that the low accuracy is due to the quality of the data and future research with other datasets could confirm or refute this. Because we do not know where these values came from aside from validating their distributions, we cannot truly expect our model accuracy to be high. Research could be performed on properly collected water samples that include measurements for our predictors. Our models could be then used and tuned for these samples to see if they can predict the water potability more accurately. Conversely, several of our predictors impact the taste of the water rather than the potability of the water. It is conceivable that the predictors used in this dataset may simply be unable to accurately classify water as potable due to the varying taste thresholds across communities. Our research did not necessarily yield the desired outcome, so there is a lot of room for growth in the future.

As previously stated, none of our datasets performed severely worse than any of the others. The only model that truly raised some concerns was AdaBoosting. This model performed about as well as the others but was tuned to only run five trees within the model to get the optimal training parameters. This is something to look out for in future research. Although the training and test results were highest with this number of trees, it is a pretty minimal use of the powerful model and could lead to some bias and misinformation. The use of other models, such as Gradient Boosting which seemed to run more as intended, may be more optimal in future research.

Although there are some ethical concerns around using this dataset to safely test water potability, this research acts as an interesting starting point to determining if these water quality predictors provide accurate predictions into water potability, and if a community tool is possible. While we do not currently plan to use this model to generalize due to safety concerns, this data is being used to practice model tuning and strengthen machine learning techniques. If data were to be ethically collected from a trustworthy source, like a governmental agency, the methods discussed in this report could potentially be extended to this kind of generalization.

Sources:

Levin, R., Villanueva, C.M., Beene, D. et al. (2023) *US drinking water quality: exposure risk profiles for seven legacy and emerging contaminants*. Journal of Exposure Science & Environmental Epidemiology. https://doi.org/10.1038/s41370-023-00597-z

US EPA. (2023). *National Primary Drinking Water Regulations | US EPA.* US EPA. https://www.epa.gov/ground-water-and-drinking-water/national-primary-drinking-water-regulations

Zhu, M., Wang, J., Yang X., et al. (2022) *A review of the application of machine learning in water quality evaluation*, Eco-Environment & Health, 1(2), 107-116. https://doi.org/10.1016/j.eehl.2022.06.001.

World Health Organization. (2022). *Guidelines for drinking-water quality* (4th ed.). World Health Organization. https://www.who.int/publications/i/item/9789240045064

Helen "Nellie" Dawson/Ryan Soucy

DSP 556

Final Project: Water Potability

```
#libraries
from google.colab import drive
drive.mount('/content/drive')
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline

import pandas as pd
import time
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV

from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression

from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import StackingClassifier
```

```
     Mounted at /content/drive
```

```
# Function to Find the Default Hyperparameter(s) of a Model
def find_parameter(model):
  model1= model()
  print("Default Parameters:")
  for param, value in model1.get_params().items():
    print(f"{param}: {value}")

# Compare while tuning then compare across
```

Data Preprocessing

```
# Import Data
water = pd.DataFrame(pd.read_csv ('/content/drive/My Drive/Colab Notebooks/DSP 556 Final Project/water_potability.csv'))
water.describe()
# water.dtypes()
```

	ph	Hardness	Solids	Chloramines	Sulfate	Conductivity	Org
count	2785.000000	3276.000000	3276.000000	3276.000000	2495.000000	3276.000000	
mean	7.080795	196.369496	22014.092526	7.122277	333.775777	426.205111	
std	1.594320	32.879761	8768.570828	1.583085	41.416840	80.824064	
min	0.000000	47.432000	320.942611	0.352000	129.000000	181.483754	
25%	6.093092	176.850538	15666.690297	6.127421	307.699498	365.734414	
50%	7.036752	196.967627	20927.833607	7.130299	333.073546	421.884968	
75%	8.062066	216.667456	27332.762127	8.114887	359.950170	481.792304	
max	14.000000	323.124000	61227.196008	13.127000	481.030642	753.342620	

```
# Dropping Variables - Solids & Organic Carbon
water = water.drop(columns = ['Solids', 'Organic_carbon'])
water.head()
```

	ph	Hardness	Chloramines	Sulfate	Conductivity	Trihalomethanes	Turbidit
0	NaN	204.890455	7.300212	368.516441	564.308654	86.990970	2.96313
1	3.716080	129.422921	6.635246	NaN	592.885359	56.329076	4.50065
2	8.099124	224.236259	9.275884	NaN	418.606213	66.420093	3.05593

There is a chance that the water quality dataset is synthesized, so we ran a check on the data to determine whether or not the variable measurements were somewhat reasonable. We did this by researching the acceptable potable water measurements for each variable. We then separated out the potable water samples from the non-potable water samples and found the mean and standard deviation of these variables. We determined that Solids and Organic Carbons were not acceptable measurements for potable water, so we removed them from the dataset.

```
# Dealing with NA Values: Remove Observations with null values
water = water.dropna()
water.describe()
```

	ph	Hardness	Chloramines	Sulfate	Conductivity	Trihalomethanes
count	2011.000000	2011.000000	2011.000000	2011.000000	2011.000000	2011.000000
mean	7.085990	195.968072	7.134338	333.224672	426.526409	66.400859
std	1.573337	32.635085	1.584820	41.205172	80.712572	16.077109
min	0.227499	73.492234	1.390871	129.000000	201.619737	8.577013
25%	6.089723	176.744938	6.138895	307.632511	366.680307	55.952664
50%	7.027297	197.191839	7.143907	332.232177	423.455906	66.542198
75%	8.052969	216.441070	8.109726	359.330555	482.373169	77.291925
max	14.000000	317.338124	13.127000	481.030642	753.342620	124.000000

Of the 3216 original water samples, we were able to retain 2011 samples that included all of the variables. We determined that this was a large enough sample size to perform our analysis with.

```
# Split based on Potability then rerun data summary
potable  = water[water["Potability"] == 1]
potable.describe()
```

	ph	Hardness	Chloramines	Sulfate	Conductivity	Trihalomethanes	Tu
count	811.000000	811.000000	811.000000	811.000000	811.000000	811.000000	81
mean	7.113791	195.908341	7.174395	332.457832	425.005423	66.581596	3
std	1.437623	35.301146	1.732796	47.446190	81.950982	16.297713	(
min	0.227499	73.492234	1.390871	129.000000	201.619737	8.577013	1
25%	6.256039	174.380497	6.106169	301.768819	360.275012	55.751069	3
50%	7.046549	197.617494	7.212254	331.087177	421.099917	66.612984	4
75%	7.955161	218.414531	8.181431	365.632984	482.296528	77.372587	4
max	11.898078	317.338124	13.127000	481.030642	695.369528	124.000000	6

```
no_potable= water[water["Potability"] == 0]
no_potable.describe()
```

```
                    ph    Hardness  Chloramines     Sulfate  Conductivity  Trihalomethanes
```

```
# Define Variables
X = water.drop(columns=['Potability'])
y = water['Potability']

# Train/Test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify = y, random_state = 1)
```

```
sclr = StandardScaler().set_output(transform = "pandas")
scaler = sclr.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

The data was scaled using the training data and fit to the test data to avoid data leakage. The data is also individually scaled using a pipeline for every GridSearch separately to avoid data leakage. Scaled data is only used on the models that will be affected by the variance of the different predictors. These models include Support Vector Classifier and Logistic Regression.

Base Models

Decision Trees

```
find_parameter(DecisionTreeClassifier)

     Default Parameters:
     ccp_alpha: 0.0
     class_weight: None
     criterion: gini
     max_depth: None
     max_features: None
     max_leaf_nodes: None
     min_impurity_decrease: 0.0
     min_samples_leaf: 1
     min_samples_split: 2
     min_weight_fraction_leaf: 0.0
     random_state: None
     splitter: best
```

Hyperparameter Selection: We choose to adjust the parameters max_depth, min_samples_split, and ccp_alpha.

max_depth - maximum depth of the tree. Normally a tree will continue until all leaves contain less than min_samples_split samples, but if we adjust the max_depth to be smaller, we are attempting to avoid overfitting to the noise at the smaller more difficult to distinguish nodes.

min_samples_split - minimum number of samples required to split an internal node This determines how much power individual points may have over the node splitting. Especially as we approach the later nodes in the tree.

ccp_alpha - prunes to avoid over-fitting Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than ccp_alpha will be chosen.

```
# Decision Trees (DTC)
model = DecisionTreeClassifier(random_state = 1)

# Define Parameter Ranges
max_depth_range = range(5, 11)
min_samples_range = range(2, 6)
ccp_alpha_range = [0, 0.001, 0.01, 0.1, 1]
param_grid = {'max_depth' : max_depth_range, 'min_samples_split' : min_samples_range, 'ccp_alpha' : ccp_alpha_range}

# Parameter Tuning via GridSearchCV
grid_search = GridSearchCV(model, param_grid, cv = 5,
                                scoring = "accuracy",
                                return_train_score = True)
grid_search.fit(X_train, y_train)
print("Best parameters: {}".format(grid_search.best_params_))
print("Mean cross-validated score: {:.3f}".format(grid_search.best_score_))

     Best parameters: {'ccp_alpha': 0.001, 'max_depth': 5, 'min_samples_split': 2}
     Mean cross-validated score: 0.644
```

```
# Best Tree
best_model = DecisionTreeClassifier(ccp_alpha = 0.001, max_depth = 5, min_samples_split = 2,random_state = 1).fit(X_train,y_train)
print("Best Tree test set score: {:.3f}".format(best_model.score(X_test, y_test)))
```

```
Best Tree test set score: 0.605
```

```
fit_time = []
score = []
train_score = []
test_score = []
params = [1, 5, 10, 15, 20, 25, 30]

for p in params:
        model = DecisionTreeClassifier(max_depth = p, random_state = 1)
        start = time.time()
        model.fit(X_train,y_train)
        stop = time.time()
        fit_time.append(stop - start)
        score.append(model.score(X_test, y_test))

print("Param, Fit Times, Scores \n")
for p, t, s in list(zip(params, fit_time, score)):
    print(f"{p:<5.0f} {t:>7.3f} {s:>10.3f}")
```

```
Param, Fit Times, Scores

1       0.007      0.618
5       0.010      0.605
10      0.014      0.605
15      0.017      0.558
20      0.019      0.551
25      0.019      0.548
30      0.020      0.548
```

DECISION TREE RESULTS: We found that with a best parameters of ccp_alpha = 0, max_depth = 12, and min_samples_split = 3, we see a test accuracy of 0.581. This is concerning because our training set was quite a bit higher, which indicates some slight overfitting. As we tested the affect max_depth has on time, we see a drop in accuracy as the max_depth increases, but the time to run the model remains relatively unchanged. This model is fast, so the runtime is not a big concern when tuning.

Support Vector Machines Classifier (SVC)

```
find_parameter(SVC)
```

```
Default Parameters:
C: 1.0
break_ties: False
cache_size: 200
class_weight: None
coef0: 0.0
decision_function_shape: ovr
degree: 3
gamma: scale
kernel: rbf
max_iter: -1
probability: False
random_state: None
shrinking: True
tol: 0.001
verbose: False
```

Hyperparameter Selection: We choose to adjust the parameter C. C is a regularization parameter which dictates how much you want to avoid misclassifying each training example

```
# Model and Preprocessing
scaler = StandardScaler()
preprocessor = ColumnTransformer([("scale", scaler, X.columns)])
model = Pipeline(steps = [("preprocess", preprocessor),("svc", SVC(random_state = 1))])

# Define Parameter Ranges
C_range =  [.01, .1, 1, 2, 3, 5]
param_grid = {'svc__C': C_range}

# Parameter Tuning via GridSearchCV
```

```
grid_search = GridSearchCV(model, param_grid, cv = 5,
                          scoring='accuracy',
                          return_train_score=True,
                          refit = True)
grid_search.fit(X_train, y_train)
print("Best parameters: {}".format(grid_search.best_params_))
print("Mean cross-validated score: {:.3f}".format(grid_search.best_score_))
```

```
Best parameters: {'svc__C': 2}
Mean cross-validated score: 0.678
```

```
# Best SVC
best_model = SVC (C = 2,random_state = 1).fit(X_train_scaled,y_train)
print("Best SVC test set score: {:.3f}".format(best_model.score(X_test_scaled, y_test)))
```

```
Best SVC test set score: 0.658
```

```
fit_time = []
score = []
train_score = []
test_score = []
params =  [.01, .1, 0.5, 1, 5, 10, 15, 20]

for p in params:
        model = SVC(C = p, random_state = 1)
        start = time.time()
        model.fit(X_train,y_train)
        stop = time.time()
        fit_time.append(stop - start)
        score.append(model.score(X_test, y_test))

print("Param, Fit Times, Scores \n")
for p, t, s in list(zip(params, fit_time, score)):
    print(f"{p:<5.2f} {t:>7.3f} {s:>10.3f}")
```

```
Param, Fit Times, Scores

0.01    0.133      0.596
0.10    0.158      0.596
0.50    0.159      0.596
1.00    0.157      0.596
5.00    0.174      0.598
10.00   0.166      0.600
15.00   0.178      0.603
20.00   0.178      0.600
```

SVC RESULTS: We found that with a best parameter of C = 2, we see a test accuracy of 0.658. Our training set was only slightly higher accuracy at .678. As we tested the affect C has on time, we see a very slight increase in accuracy and time.

Random Forest

```
from sklearn.ensemble import RandomForestClassifier

find_parameter(RandomForestClassifier)
```

```
Default Parameters:
bootstrap: True
ccp_alpha: 0.0
class_weight: None
criterion: gini
max_depth: None
max_features: sqrt
max_leaf_nodes: None
max_samples: None
min_impurity_decrease: 0.0
min_samples_leaf: 1
min_samples_split: 2
min_weight_fraction_leaf: 0.0
n_estimators: 100
n_jobs: None
oob_score: False
random_state: None
verbose: 0
warm_start: False
```

Hyperparameter Selection: We choose to adjust the parameters max_depth, min_samples_split, and n_estimators

We will not adjust ccp_alpha, as we found it not to be as powerful of a parameter of the others when exploring Decision Trees. It gives us the opportunity to adjust the parameter that Random Forest introduces called number of estimators which determines how many decision trees are in the Random Forest Model.

n_estimators - the number of Decision Trees built within each forest.

PARAMETERS ON SMALLER DECISION TREES: max_depth - maximum depth of the tree. min_samples_split - minimum number of samples required to split an internal node

```
# Random Forest (RFC)
model = RandomForestClassifier(random_state = 1)

# Define Parameter Ranges
max_depth_range = [3, 5, 7, 10]
min_samples_range = [2, 3, 5]
estimators_range = [100, 300, 500]
param_grid = { 'max_depth' : max_depth_range,
              'min_samples_split' : min_samples_range,
              'n_estimators' : estimators_range}

# Parameter Tuning via GridSearchCV
grid_search = GridSearchCV(model, param_grid, cv = 5,
                          scoring='accuracy',
                          return_train_score=True,
                          refit = True)
grid_search.fit(X_train, y_train)
print("Best parameters: {}".format(grid_search.best_params_))
print("Mean cross-validated score: {:.3f}".format(grid_search.best_score_))
```

```
Best parameters: {'max_depth': 10, 'min_samples_split': 2, 'n_estimators': 500}
Mean cross-validated score: 0.678
```

```
# Best Random Forest
best_model = RandomForestClassifier(n_estimators = 500, max_depth = 10, min_samples_split = 2, random_state = 1).fit(X_train, y_train)
print("Best Random Forest test set score: {:.3f}".format(best_model.score(X_test, y_test)))
```

```
Best Random Forest test set score: 0.635
```

```
fit_time = []
score = []
train_score = []
test_score = []
params =  [2, 3, 5, 7, 10, 13, 15]

for p in params:
        model = RandomForestClassifier(max_depth = p, random_state = 1)
        start = time.time()
        model.fit(X_train,y_train)
        stop = time.time()
        fit_time.append(stop - start)
        score.append(model.score(X_test, y_test))

print("Param, Fit Times, Scores \n")
for p, t, s in list(zip(params, fit_time, score)):
    print(f"{p:<5.0f} {t:>7.3f} {s:>10.3f}")
```

```
Param, Fit Times, Scores

2       0.222      0.610
3       0.233      0.618
5       0.298      0.643
7       0.353      0.640
10      0.410      0.643
13      0.451      0.638
15      0.491      0.633
```

```
fit_time = []
score = []
train_score = []
test_score = []
params =  [100, 200, 300, 400, 500, 600]

for p in params:
        model = RandomForestClassifier(n_estimators = p, random_state = 1)
        start = time.time()
        model.fit(X_train,y_train)
        stop = time.time()
        fit_time.append(stop - start)
        score.append(model.score(X_test, y_test))

print("Param, Fit Times, Scores \n")
for p, t, s in list(zip(params, fit_time, score)):
    print(f"{p:<5.0f} {t:>7.3f} {s:>10.3f}")
```

```
Param, Fit Times, Scores

100     0.522      0.633
200     1.126      0.625
300     2.262      0.625
400     2.980      0.628
500     2.611      0.633
600     3.084      0.640
```

RANDOM FOREST RESULTS: We found that with a best parameters of n_estimators = 500, max_depth = 15, and min_samples_split = 5, we see a test accuracy of 0.628. Although these scores are not exceptional, this Random Forest Model performs slightly better than the more simple Decision Tree Classifier, which had a score of .581. The max_depth of each tree within the random forest seems to have a significant impact on the accuracy and not as much so on the fit times, while the number of estimators has the opposite of large fit time increases with not drastic changes in accuracy.

Logistic Regression

```
find_parameter(LogisticRegression)
```

```
Default Parameters:
C: 1.0
class_weight: None
dual: False
fit_intercept: True
intercept_scaling: 1
l1_ratio: None
max_iter: 100
multi_class: auto
n_jobs: None
penalty: l2
random_state: None
solver: lbfgs
tol: 0.0001
verbose: 0
warm_start: False
```

HYPERPARAMETER SELECTION:

C - regularization strength

Class weights stays default because we scaled data

```
# Model and Preprocessing
scaler = StandardScaler()
preprocessor = ColumnTransformer([("scale", scaler, X.columns)])
model = Pipeline(steps = [("preprocess", preprocessor),("LR", LogisticRegression(random_state = 1))])


# Define Parameter Ranges
C_range =  [.01, .1, 1, 2]
param_grid = {'LR__C': C_range}

# Parameter Tuning via GridSearchCV
grid_search = GridSearchCV(model, param_grid, cv = 5,
                          scoring='accuracy',
                          return_train_score=True)
grid_search.fit(X_train, y_train)
print("Best parameters: {}".format(grid_search.best_params_))
print("Mean cross-validated score: {:.3f}".format(grid_search.best_score_))
```

```
Best parameters: {'LR__C': 0.01}
Mean cross-validated score: 0.597
```

```
# Best Logistic Regression
best_model = LogisticRegression(C = 0.01, random_state = 1).fit(X_train_scaled,y_train)
print("Best Logistic Regression test set score: {:.3f}".format(best_model.score(X_test_scaled, y_test)))
```

```
Best Logistic Regression test set score: 0.596
```

```
fit_time = []
score = []
train_score = []
test_score = []
params =  [.01, .1, 0.5, 1, 5, 10, 15, 20]

for p in params:
        model = LogisticRegression(C = p, random_state = 1)
        start = time.time()
        model.fit(X_train,y_train)
        stop = time.time()
        fit_time.append(stop - start)
        score.append(model.score(X_test, y_test))

print("Param, Fit Times, Scores \n")
for p, t, s in list(zip(params, fit_time, score)):
    print(f"{p:<5.2f} {t:>7.3f} {s:>10.3f}")
```

```
Param, Fit Times, Scores

0.01    0.037      0.596
0.10    0.019      0.596
0.50    0.031      0.596
1.00    0.021      0.596
5.00    0.020      0.596
10.00   0.024      0.596
15.00   0.017      0.596
20.00   0.023      0.596
```

LOGISTIC REGRESSION RESULTS:

Ensemble Models

SVC Best Params: C = 2

Random Forest Best Params:

- max_depth = 12
- min_samples_split = 3
- n_estimators = 300

Compare Ensemble Models to Base Learners

Bagging Classifier

```
find_parameter(BaggingClassifier)
```

```
Default Parameters:
base_estimator: deprecated
bootstrap: True
bootstrap_features: False
estimator: None
max_features: 1.0
max_samples: 1.0
n_estimators: 10
n_jobs: None
oob_score: False
random_state: None
verbose: 0
warm_start: False
```

Hyperparameter Selection: max_features - number of features selected from the numerical predictors/parameters

max_samples - maximum number of samples from X used to train the base decision trees

n_estimators - number of decision trees ran

```
fit_time = []
score = []
params = [10, 50, 100, 200, 300, 400, 500]
model = BaggingClassifier

for p in params:
    model1 = model(n_estimators = p, max_samples = p, max_features = 7,random_state = 1)
    start = time.time()
    model1.fit(X_train, y_train)
    stop = time.time()
    fit_time.append(stop - start)
    score.append(model1.score(X_test, y_test))

print("Params Fit times Score \n")
for p, t, s in list(zip(params, fit_time, score)):
    print(f"{p:<5} {t:>10.1f}  {s:>5.4f}")
```

```
Params Fit times Score

10           0.0  0.4640
50           0.1  0.6129
100          0.2  0.6303
200          0.6  0.6303
300          1.1  0.6427
400          2.1  0.6476
500          3.9  0.6303
```

We determined that the maximum features should remain at 7 because this resulted in the most accuracy, we do not have an absurd amount of features in the data, and the run times were still quite low.

The cross validated accuracy of the n_estimator and max_samples kept wanting to increase. The higher that these numbers are, though, the longer the fit times. We determined that the accuracy of the model started to level off when these values were around 200, while the fit time for 500 was almost 3x that of 200. This helped determine the ranges below.

```
model = BaggingClassifier(random_state = 1)

# Define Parameters
max_features_range = [7]
max_samples_range = [50, 100, 200]
n_estimators_range = [50, 100, 200]
param_grid = {'max_features': max_features_range,
              'max_samples': max_samples_range,
              'n_estimators': n_estimators_range}

# Parameter Tuning via GridSearchCV
grid_search = GridSearchCV(model, param_grid, cv = 5,
                          scoring='accuracy',
                          return_train_score=True,
                          refit = True)
grid_search.fit(X_train, y_train)
print("Best parameters: {}".format(grid_search.best_params_))
print("Mean cross-validated score: {:.3f}".format(grid_search.best_score_))


# Best Model
best_model = BaggingClassifier(**grid_search.best_params_, random_state = 1).fit(X_train,y_train)
print("Best test set score: {:.3f}".format(best_model.score(X_test, y_test)))
```

```
Best parameters: {'max_features': 7, 'max_samples': 200, 'n_estimators': 200}
Mean cross-validated score: 0.669
Best test set score: 0.630
```

BAGGING CLASSIFIER RESULTS: We found that with a best parameters of max_feature = 7, max_samples = 200, and n_estimator = 200, we see a test score of 0.630. Although these scores are not exceptional, model performs slightly better than the more simple Decision Tree Classifier, which had a score of .581 and comparable to Random forest which had a test score of .628. By Bagging the Decision Tree Classifier, we hoped to test the data on a larger randomized population to determine the test accuracy. This showed us that our model accuracy is still quite low, but is avoiding overfitting and underfitting because the training and test scores are quite equal.

AdaBoosting Classifier

```
find_parameter(AdaBoostClassifier)
```

```
Default Parameters:
algorithm: SAMME.R
base_estimator: deprecated
estimator: None
learning_rate: 1.0
n_estimators: 50
random_state: None
```

Hyperparameter selection:

learning_rate - Weight applied to each classifier at each boosting iteration. A higher learning rate increases the contribution of each classifier. There is a trade-off between the learning_rate and n_estimators parameters

n_estimators - The number of decision trees run in the model

```
model = AdaBoostClassifier(DecisionTreeClassifier(random_state = 1), random_state = 1)

# Define Parameters
learning_range = [0.001, 0.01, 0.1, 1]
estimators_range = [5, 10, 25, 50, 100]
param_grid = {'learning_rate': learning_range,
              'n_estimators': estimators_range}

# Parameter Tuning via GridSearchCV
grid_search = GridSearchCV(model, param_grid = param_grid,
                                cv = 5,
                                scoring='accuracy',
                                return_train_score=True,
                                refit = True)
grid_search.fit(X_train, y_train)
print("Best parameters: {}".format(grid_search.best_params_))
print("Mean cross-validated score: {:.3f}".format(grid_search.best_score_))


# Best AdaBoostClassifier
best_model = AdaBoostClassifier(**grid_search.best_params_, random_state = 1).fit(X_train,y_train)
print("Best test set score: {:.3f}".format(best_model.score(X_test, y_test)))
```

```
Best parameters: {'learning_rate': 0.001, 'n_estimators': 5}
Mean cross-validated score: 0.594
Best test set score: 0.618
```

```
fit_time = []
score = []
params = [5, 10, 25, 50, 100]
model = AdaBoostClassifier

for p in params:
    model1 = model(n_estimators = p, random_state = 1)
    start = time.time()
    model1.fit(X_train, y_train)
    stop = time.time()
    fit_time.append(stop - start)
    score.append(model1.score(X_test, y_test))

print("Params Fit times Score \n")
for p, t, s in list(zip(params, fit_time, score)):
    print(f"{p:<5} {t:>10.1f}  {s:>5.4f}")
```

```
Params Fit times Score

5            0.0  0.6079
10           0.0  0.5931
25           0.1  0.5757
50           0.2  0.5931
100          0.4  0.5881
```

ADABOOST CLASSIFIER RESULTS: We found that with the best parameters of learning_rate = .001 and n_estimators = 5, we see a test score of 0.618. This model performs about as well on the test set as the Bagging model. We are a bit concerned that this model is potentially underfitting the data or, more likely, not running enough decision trees to decrease bias and variance due to the n_estimators only being 5. This is the optimal parameter, but it just does not seem to be maximizing ADA Boosting's potential, so it feels worth noting.

Gradient Boosting

```
find_parameter(GradientBoostingClassifier)
```

```
Default Parameters:
ccp_alpha: 0.0
criterion: friedman_mse
init: None
learning_rate: 0.1
loss: log_loss
max_depth: 3
max_features: None
max_leaf_nodes: None
min_impurity_decrease: 0.0
min_samples_leaf: 1
min_samples_split: 2
min_weight_fraction_leaf: 0.0
```

```
n_estimators: 100
n_iter_no_change: None
random_state: None
subsample: 1.0
tol: 0.0001
validation_fraction: 0.1
verbose: 0
warm_start: False
```

Hyperparameter Selection:

learning_rate - Weight applied to each classifier at each boosting iteration. A higher learning rate increases the contribution of each classifier. There is a trade-off between the learning_rate and n_estimators parameters

n_estimators - The number of decision trees run in the model

max_depth - the maximum depth of the trees run

```
model = GradientBoostingClassifier(random_state = 1)

# Define Parameters
learning_range = [0.01, 0.1, 1]
estimators_range = [10, 100, 200]
depth_range = [1, 5, 10]

param_grid = {'learning_rate': learning_range,
              'n_estimators': estimators_range,
              'max_depth' : depth_range}

# Parameter Tuning via GridSearchCV
grid_search = GridSearchCV(model, param_grid = param_grid,
                                cv = 5,
                                scoring = 'accuracy',
                                return_train_score = True,
                                refit = True)
grid_search.fit(X_train, y_train)
print("Best parameters: {}".format(grid_search.best_params_))
print("Mean cross-validated score: {:.3f}".format(grid_search.best_score_))

# Best AdaBoostClassifier
best_model = GradientBoostingClassifier(**grid_search.best_params_, random_state = 1).fit(X_train,y_train)
print("Best test set score: {:.3f}".format(best_model.score(X_test, y_test)))
```

```
Best parameters: {'learning_rate': 0.01, 'max_depth': 5, 'n_estimators': 200}
Mean cross-validated score: 0.666
Best test set score: 0.613
```

```
fit_time = []
score = []
params = [10, 100, 200]
model = GradientBoostingClassifier

for p in params:
    model1 = model(n_estimators = p, random_state = 1)
    start = time.time()
    model1.fit(X_train, y_train)
    stop = time.time()
    fit_time.append(stop - start)
    score.append(model1.score(X_test, y_test))

print("Params Fit times Score \n")
for p, t, s in list(zip(params, fit_time, score)):
    print(f"{p:<5} {t:>10.1f}  {s:>5.4f}")
```

```
Params Fit times Score

10           0.1  0.6278
100          0.5  0.6377
200          1.0  0.6476
```

```
fit_time = []
score = []
params = [1, 5, 10]
model = GradientBoostingClassifier

for p in params:
    model1 = model(max_depth = p, random_state = 1)
    start = time.time()
    model1.fit(X_train, y_train)
    stop = time.time()
    fit_time.append(stop - start)
    score.append(model1.score(X_test, y_test))

print("Params Fit times Score \n")
for p, t, s in list(zip(params, fit_time, score)):
    print(f"{p:<5} {t:>10.1f}  {s:>5.4f}")
```

```
Params Fit times Score

1            0.2  0.6030
5            1.1  0.6203
10           2.6  0.6179
```

GRADIENT BOOST CLASSIFIER RESULTS:

We found that with the best parameters of learning_rate = .01, max_depth = 5, and n_estimators = 200, we see a test score of 0.613. This model performs about as well on the test set as the Bagging model and ADA Boosting model. The fit times of these models were all pretty low, but the max_depth did seem to have the most impact of a few seconds, which is pretty minimal. These tuned parameters also seem to be more reasonable than ADA Boosting's. Having relatively shallow trees with a max_depth of 5 may be underfitting the data on each individual tree, but it saves time in the model and the n_estimators of 200 decreases the impact of this underfitting by generating 200 trees within the model. This is exactly how the model is supposed to run.

Stacking

```
stk = StackingClassifier(estimators= [
  ('model1', DecisionTreeClassifier(ccp_alpha = 0.001, max_depth = 5, min_samples_split = 2)),
  ('model2', LogisticRegression(C = 0.01)),
],
  final_estimator= SVC(random_state = 1)
)
print("Default Parameters:")
for param, value in stk.get_params().items():
  print(f"{param}: {value}")
```

```
Default Parameters:
cv: None
estimators: [('model1', DecisionTreeClassifier(ccp_alpha=0.001, max_depth=5)), ('model2', LogisticRegression(C=0.01))]
final_estimator__C: 1.0
final_estimator__break_ties: False
final_estimator__cache_size: 200
final_estimator__class_weight: None
final_estimator__coef0: 0.0
final_estimator__decision_function_shape: ovr
final_estimator__degree: 3
final_estimator__gamma: scale
final_estimator__kernel: rbf
final_estimator__max_iter: -1
final_estimator__probability: False
final_estimator__random_state: 1
final_estimator__shrinking: True
final_estimator__tol: 0.001
final_estimator__verbose: False
final_estimator: SVC(random_state=1)
n_jobs: None
passthrough: False
stack_method: auto
verbose: 0
model1: DecisionTreeClassifier(ccp_alpha=0.001, max_depth=5)
model2: LogisticRegression(C=0.01)
model1__ccp_alpha: 0.001
model1__class_weight: None
model1__criterion: gini
model1__max_depth: 5
model1__max_features: None
model1__max_leaf_nodes: None
model1__min_impurity_decrease: 0.0
```

```
 model1__min_samples_leaf: 1
 model1__min_samples_split: 2
 model1__min_weight_fraction_leaf: 0.0
 model1__random_state: None
 model1__splitter: best
 model2__C: 0.01
 model2__class_weight: None
 model2__dual: False
 model2__fit_intercept: True
 model2__intercept_scaling: 1
 model2__l1_ratio: None
 model2__max_iter: 100
 model2__multi_class: auto
 model2__n_jobs: None
 model2__penalty: l2
 model2__random_state: None
 model2__solver: lbfgs
 model2__tol: 0.0001
 model2__verbose: 0
 model2__warm_start: False
```

Hyperparameter Selection:

Base learners: models property tuned using parameters determined in Base Models above

DecisionTreeClassifier

LogisticRegression

Final Estimator:

SVC - C being re-tuned to be optimal within this model. C is a regularization parameter which dictates how much you want to avoid misclassifying each training example

```
import time
fit_time = []
score = []
params = [.01, .1, 1, 2, 3, 5]

for p in params:
    stk = StackingClassifier(estimators= [
        ('model1', DecisionTreeClassifier(ccp_alpha = 0.001, max_depth = 5, min_samples_split = 2)),
         ('model2', LogisticRegression(C = 0.01)),
    ],
                             final_estimator= SVC(C = p, random_state = 1)
    )
    start = time.time()
    stk.fit(X_train_scaled, y_train)
    stop = time.time()
    fit_time.append(stop - start)
    score.append(stk.score(X_test_scaled, y_test))

print("Best Values fit times score \n")
for p, t, s in list(zip(params, fit_time, score)):
    print(f"{p:<5} {t:>10.1f}  {s:>5.4f}")
```

```
 Best Values fit times score
```