Prediction of Wine Quality

Definition

Project Overview

In the world of enology, what appears on the surface to be a simple process becomes much more complex as the chemistry of wine making unfolds itself. Individuals have gone to great lengths in order to categorize and predict the profile and quality of a given wine. A small list of individuals, for instance, bring home six figure pay days just by their ability to categorize, rate, and recommend wines. Even for daily consumers, the quality of a wine from a measurable standard is helpful for either personal enjoyment or hosting a successful dinner party. Historically it has been shown that it is possible to attack this problem from the point of machine learning via work done by Paulo Cortez and colleagues.

The wine quality data set found on the UCI Machine Learning Repository was donated in October 2009 and provides the inputs and desired output needed to show how a machine learning algorithm can help us to accurately predict a wine's quality based on its chemistry. The dataset contains 4898 observations which include 11 physiochemical variables outlined below.

1. Volatile acidity
2. Citric acid
3. Residual sugar
4. Chlorides
5. Free sulfur dioxide
6. Density
7. pH
8. sulphates
9. alcohol
10. fixed acidity
11. quality

Using machine learning, we may be able to use this data to provide a quicker, cheaper option for individuals or wineries to predict their wine based off its chemical properties.

Problem Statement

The problem: How do we use machine learning to create an algorithm that will predict the quality of wine on a numeric scale given its physiochemical properties?

A solution to the problem of wine quality prediction can be approached using feature selection and a multi model testing approach. In this particular dataset we have complete data, so we do not not need to deal with the missingness of features.

1. Cortez, Paulo et. Al. “Modeling Wine Preferences by Data Mining from Physiochemical Properties.” https://www.sciencedirect.com/science/article/pii/S0167923609001377
2. UCI website library: http://archive.ics.uci.edu/ml/datasets/Wine+Quality

To begin with, since we are unsure of the relevance of each feature we can explore the data to find features with the most variance and compare this with the features that are most associated with the outcomes of interest. We will also look at the collinearity between our inputs in order to choose the best features for our problem.

To go through this process, we can start with normalizing the data and then employing principal component analysis. We will look at each principal component and see the eigenvalues of our inputs to look for variance. We can also look at the collinearity of our data to remove redundant inputs using a correlation matrix; if two are highly correlated with one another we can look into which one is most associated with our output and choose accordingly. A good approach is to use a Lasso regression which will punish the coefficients of variables least associated with wine quality.

After the above, we can combine our observations to choose an optimized set of inputs. Finally, we can experiment with feature engineering to find groupings of wine that are the most associated with one another using a K-nearest neighbors clustering approach to get the final dataset we can use to solve the problem.

Once our final candidates are chosen, we can test a few different models out to see which one performs the best given the nature of our data. Each model can be put into a pipeline and cross validated amongst every combination of its chosen set of hyperparameters on a test portion of the data. To choose a winner we will have to test our given metric on an unlooked at test set.

Metrics

As stated above, the metric we choose must be appropriate to solve the problem at hand. There are two factors I am considering when choosing the metrics to look at. We need to know how close our predictions reflect actual outputs generally, but we also need to think about scenarios that do not show up as much.

The first metric tested will be the mean absolute error (MAE). As the name suggests, the MAE will find the absolute distance between each prediction and the actual value of a wine's quality, averaging across all predictions. This will allow us to see what the error rate of our predictions are generally.

The second metric we evaluate will be accuracy within a tolerance threshold. Since not every quality score is given at the same frequency, this will help us to see how well we perform on outlier quality scores that do not show up as frequently better than just the average error. An example is provided below (we can change the threshold and report on our outcomes multiple times). This can be run for each possible score.

A wine quality score of 3 is seen in the data. A threshold (T=1) will be used to compute accuracy for all predictions of the actual score. For instance, a prediction of 4 or 2 would be counted as a correct response using this threshold, while a score of 4.01 or 1.99 would not.

Analysis

Data Exploration & Exploratory Visualization

We will begin by importing general libraries we will need in order to manipulate our data, explore the data, and visualize the data. We will of course also need to pull in our dataframe we will be using throughout the analysis.

Figure 1: Example of wine physiochemical data

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.0	0.27	0.36	20.7	0.045	45.0	170.0	1.0010	3.00	0.45	8.8	6
1	6.3	0.30	0.34	1.6	0.049	14.0	132.0	0.9940	3.30	0.49	9.5	6
2	8.1	0.28	0.40	6.9	0.050	30.0	97.0	0.9951	3.26	0.44	10.1	6
3	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.9956	3.19	0.40	9.9	6
4	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.9956	3.19	0.40	9.9	6

Figure 2: General Statistics of Wine Elements

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
count	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000
mean	6.854788	0.278241	0.334192	6.391415	0.045772	35.308085	138.360657	0.994027	3.188267	0.489847	10.514267	5.877909
std	0.843868	0.100795	0.121020	5.072058	0.021848	17.007137	42.498065	0.002991	0.151001	0.114126	1.230621	0.885639
min	3.800000	0.080000	0.000000	0.600000	0.009000	2.000000	9.000000	0.987110	2.720000	0.220000	8.000000	3.000000
25%	6.300000	0.210000	0.270000	1.700000	0.036000	23.000000	108.000000	0.991723	3.090000	0.410000	9.500000	5.000000
50%	6.800000	0.260000	0.320000	5.200000	0.043000	34.000000	134.000000	0.993740	3.180000	0.470000	10.400000	6.000000
75%	7.300000	0.320000	0.390000	9.900000	0.050000	46.000000	167.000000	0.996100	3.280000	0.550000	11.400000	6.000000
max	14.200000	1.100000	1.660000	65.800000	0.346000	289.000000	440.000000	1.038980	3.820000	1.080000	14.200000	9.000000

We can clearly see our input variables and target variable of quality.

The first thing to notice is that the variables have different scales. We will need to handle this later when performing exploratory methods and machine learning algorithms. We can also see that for our output, no score of less than 3 has ever been given for a wine's quality and never more than 9.

Three visualizations might help us to look further into the data.

1. A pairplot can show us general data distributions as well as early indicators of variable correlations.

2. Google's facets overview is a great tool to dive deeply to interactively display each variable's distibution as well as a few quick statistics to build on the higher level seaborn plot.

3. We can create an annotated correlation matrix to clearly discern possible collinear variables we may want to delve further into.

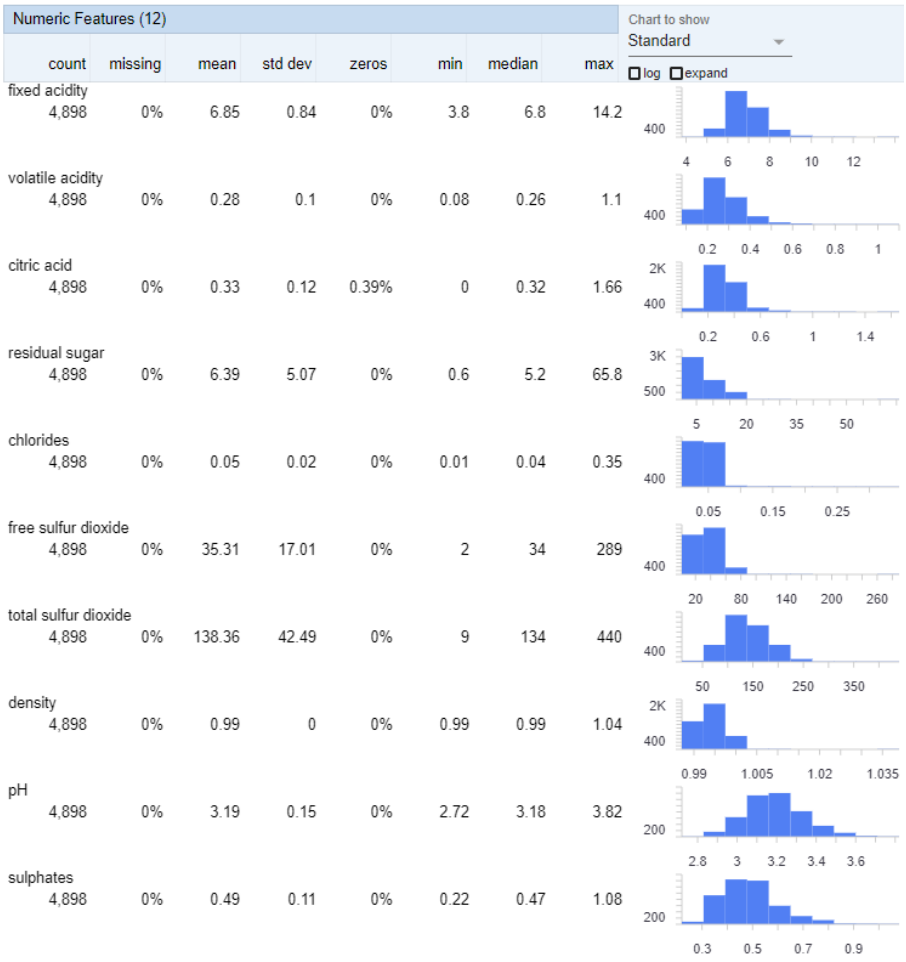
We can confirm with our pairplot that just a high level overview shows that the distribution of quality scores is varied; most scores are around the median of 6 and there are a few outliers for the highest and lowest scores. Correlations between residual sugar and density are apparent. It also appears that density is correlated strongly with alcohol.

Some variables appear to have a somewhat normal distribution such as pH; some others such as residual sugar have a Poisson (right skew) to them. This is evidence that we may need to normalize our data as well as scale it.

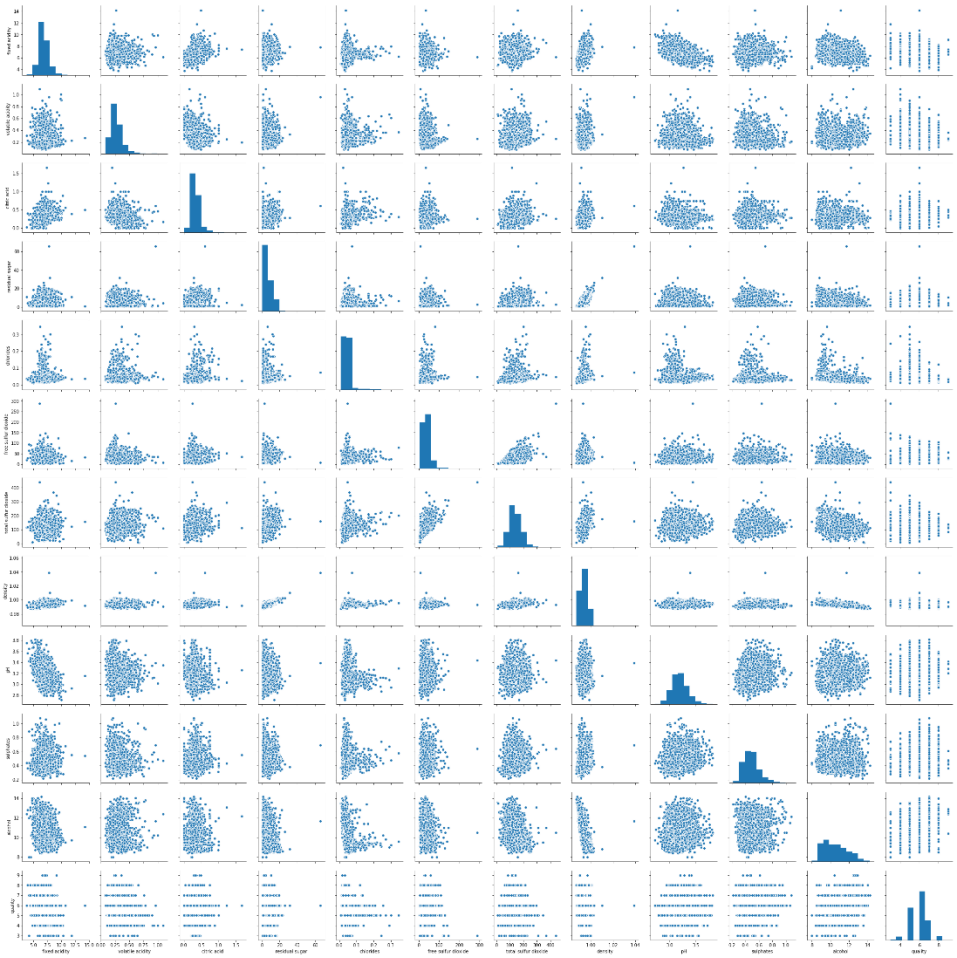
We can expand our data, log transform it, or search for specific variables. We can also look at each graph interactively. This gives us a much more robust insight into our data. It is good to note that we have complete data! We can also see that there are very few scores provided for the lowest and highest end of the spectrum for our quality output variable.

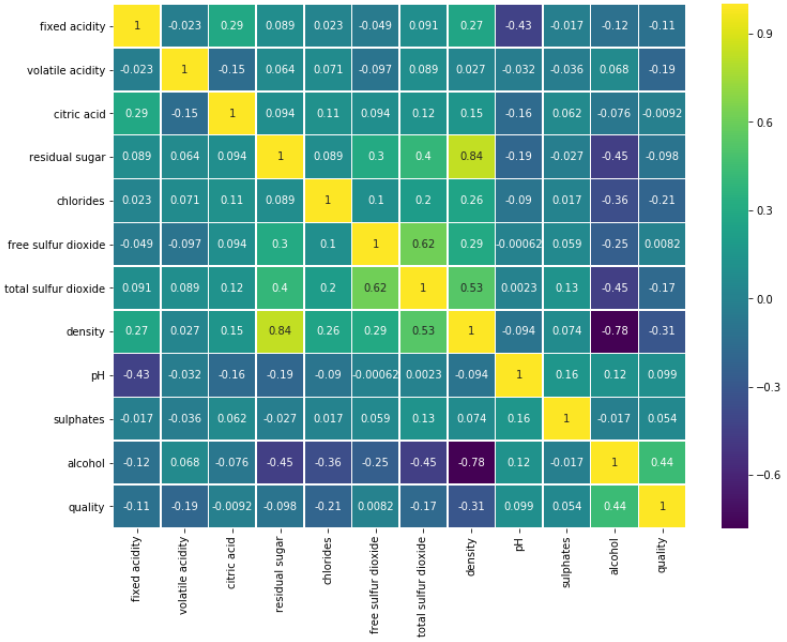
One thing that would be helpful is to delve more explicitly into variable correlations. Looking into our matrix, it looks like we need to keep an eye on density; it is highly correlated with both alcohol and residual sugar. The two sulfur dioxide variables have a medium amount of correlation, and the rest of the variables look like they are not very correlated with one another.

Figure 3: Google facet grid overview: https://pair-code.github.io/facets/



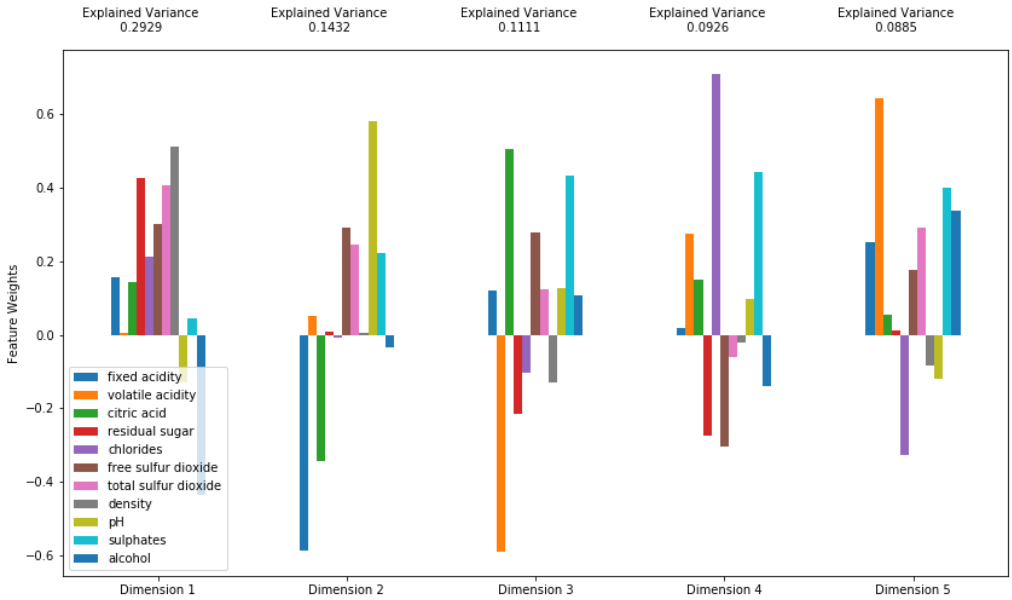
Figures 4&5: Pairplot of our data; correlation matrix of our outputs from Seaborn: https://seaborn.pydata.org/





We can dive further into the understanding of the variance in our data by running a principal component analysis. Let's first make sure we normalize our data so the eigenvalues of our output are correct. We can try to find which variables have the highest weighted contribution to the data's variance for our inputs. This can be achieved using sklearn, a machine learning library that can help to build both supervised and unsupervised models as well as preprocess data (such as the normalization performed in the outputs seen below).

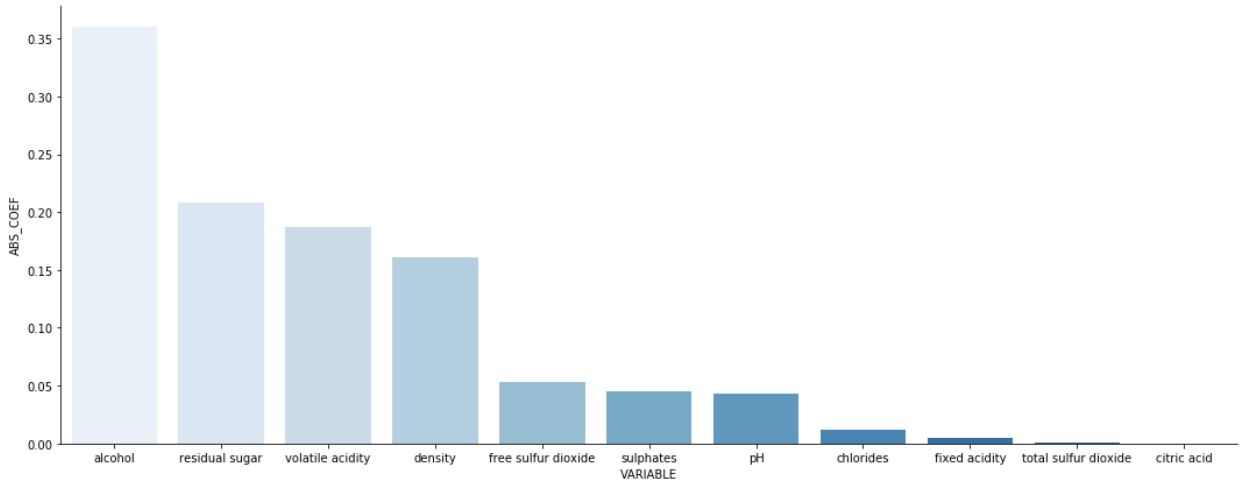
Figure 6: Principal Component Analysis of our data based on Udacity ML Nanodegree unsupervised learning section graph code



If we take the weighted average of the absolute value of all the variances by the top 5 dimensions broken out by their contributions to the overall variance (i.e. dimension 2 is 0.43 - 0.29 etc.) we can see that the variation in some variables is multiples higher vs others. We may want to consider removing some of the lowest, but to test this further beyond just variance in the data we can run a lasso regression in sklearn. Just because a variable has high variance does not mean it has a high relationship to our output of interest.

Running the regression will show the relationships of our inputs to our output. Based off this we can start to play with variable removal before any additional feature engineering. To make sure we perform the regression correctly, we first use the standardscaler class in sklearn to standardize out data. We will also have to choose our alpha level which shows how much we are punishing coefficients that are less associated with our outcome of interest.

Figure 7: Lasso Regression Results from Seaborn: https://seaborn.pydata.org/



Algorithms and Techniques

Now we are beginning to see the fruits of our explorations. The ways we have looked into the data will help us to transform our input data to choose the optimal features. After this we can practice K-Means Clustering to engineer a new feature based on clusters of our data.

We will normalize and standardize our data prior to building models; this will allow faster convergence and is also necessary to get the right outputs in some of the algorithms we will choose (such as the multiple regression).

Next we can test a pipeline of supervised models that are tuned on their hyperparameters. We will test an AdaBoost, Support Vector Regressor, Multiple Regression, and Neural Network approach. We can tune the hyperparameters of each in the pipeline and cross validate them on a training set of data in order to pick a winner.

Benchmark

Prior to any actual data preprocessing, we can test a benchmark model, a support vector machine. The paper by Cortez and colleagues shows that this is one way to measure our desired outcome. The benchmark will be tested on its absolute mean error as well as its accuracy at a tolerance of 1.0. From there we can begin processing and refining the data we will use to test quality. The benchmark model has an MAE of 0.55 and an accuracy within tolerance of 82.9%.

Analysis

Data Preprocessing

Now we can use our baseline model and proposed techniques to implement a solution to our problem. As discussed above, we can start by processing our data. The insights we gained from our correlation matrix, Lasso regression, and PCA will help us to choose a final set of candidate input variables.

Both the PCA and our lasso tests show us that we may be able to drop fixed citric acid and fixed acidity as features since they have small coefficients in the lasso as well as small variance in the PCA. We can test this with a simple support vector machine since it will be one of our candidates later and since it's a baseline model for comparison. We can also play with removing total sulfur dioxide since it comes out even lower than fixed acidity in our Lasso output. We can also try chlorides for a similar reason in the lasso outputs. Finally, our correlation matrix may be useful since density is a large correlate to both alcohol and residual sugar. The table below shows a few iterations of variable removal and the resulting MAE.

Variable Removal	**MAE**
Remove nothing	0.5528
Remove citric acid	0.5509
Remove f.a. and c.a	0.5542
Remove f.a	0.5568
Remove t.s.d	0.5652
Remove chlorides	0.5504
Remove chlrides / c.a	0.5509
Remove density, chlor, ca	0.5452

At the end it doesn't seem to hurt our absolute error in many of our testing scenarios, but the goal currently is going to be minimizing error. Interestingly we can get rid of density due to its covariance with other metrics, chloride, and citric acid to improve performance on our baseline model! We have cut down on correlations on our inputs and have also lessened the absolute mean error.

Now we will engineer a variable using KMeans clustering to see if we can improve our model. We should check which number of clusters give us silhouette scores that are passable before choosing. Choosing the number of clusters can be challenging; we want to make sure we are getting enough clusters to create differentiation with our model, but we also want to make sure that the boundaries are as clearly defined as possible. This is sometimes more of an art than a science to a degree, and we are also assuming that the variance of each distribution is spherical and has the same variance. At first glance my clusters were not to my expectations, and this enlightened me to the fact that I hadn't properly scaled my data.

Since we want to have more variability in our cluster assignment, we will use 4 clusters and see if we can improve our accuracy. As explained earlier, we also need to normalize our data for the clusters to get the right interpretation. Our unoptimized result now shows an MAE of 0.54 which is an improvement.

Implementation & Refinement

Now that we have selected the most important features, engineered a new helpful feature, and discussed the need for scaling/normalizing our data, it's time to create a pipeline to test models against one another.

We will test a SVR, AdaBoostRegressor, Lasso, and MLPRegressor against one another. We will build the pipeline to both run the models iteratively and to tune their cross validated hyperparameters. Tuning the hyperparameters will help us to refine our baseline models into the best candidates amongst parameters we choose. After this we can play with the size of our test data to come to a final solution. The right pipeline can be a challenging feat to accomplish. Each model has to be specified to the hyperparameters we wish to iterate over, and we also have to make an educated hypothesis about the right level of cross validation which can often depend on the size of our data. Below is an example – for those not familiar with python it can still be clear that the top portion of our pipeline chooses our models, the second part chooses the parameters for each model, and the

bottom ties this all together with 3 fold cross validation and the mean absolute error as our chosen measurement to pick the winning model and hyperparameters.

```
def grid_search():
    pipeline1 = Pipeline((
    ('clf', AdaBoostRegressor()),
    ))

    pipeline2 = Pipeline((
    ('clf', Lasso()),
    ))

    pipeline3 = Pipeline((
    ('clf', SVR()),
    ))

    pipeline4 = Pipeline((
    ('clf', MLPRegressor()),
    ))

    parameters1 = {
    'clf__learning_rate' :   [0.1, 0.001, 1],
    'clf__n_estimators': [10, 20, 30, 40, 50]
    }
    parameters2 = {
    'clf__alpha' :  [0.001, 1, 1]
    }
    parameters3 = {
    'clf__C': [0.1, 1.0],
    'clf__kernel': ['rbf', 'poly'],
    'clf__gamma': [0.01, 0.1, 1.0],
    }
    parameters4 = {
    'clf__hidden_layer_sizes': [i for i in range(1,5)],
    'clf__solver' : ['lbfgs', 'adam']
    }
    pars = [parameters1, parameters2, parameters3, parameters4]
    pips = [pipeline1, pipeline2, pipeline3, pipeline4]

    print ("starting Gridsearch")
    for i in range(len(pars)):
        gs = GridSearchCV(pips[i], pars[i], cv =3, scoring= 'neg_mean_absolute_error')
        gs = gs.fit(X_train, y_train)
        print ("finished Gridsearch")
        print (gs.best_score_)
        print (gs.get_params)
```

models

hyperparameters

Cross validation and scoring (MAE)

But why choose these models in the first place? Let's dig a little deeper into our chosen models. The Support Vector Machine can come into handy when we need to identify complex relationships in our data that are beyond straight lines. Depending on what kernel we use, we can create different functions that will try to best cut our data along a well-defined boundary. Most importantly, what sets a SVR apart is the fact that it utilizes the kernel trick. Stated simply, this will allow an unravelling of our data, creating a dataset that may not be linearly separable in our n dimension space, but may in fact be so in a higher dimensional space. We will try different kernel families which will serve as the basis upon which we create data in a new state space. The "C" value will serve as our parameter of error when finding values inside our defined boundaries. The gamma we choose will be the coefficient assigned to our kernel. One difficulty in the implementation above is the notably long training time it takes for the SVR models versus some of our other choices.

The Adaboost Regressor is an ensemble method. The algorithm will take a base regressor (default in our example is the decision tree) and fit to our data. After this, additional copies will be fit on our classifier but adjust towards instances of data that are proving most difficult to correctly predict with the least error. Thus, we see training sets with more difficult examples with each new fit. The number of estimators is what changes the

number of fits in the ensemble in our example, while the learning rate trades off with the number of estimators by shrinking the contributions of new models that are added to the ensemble series.

The Lasso Regression, described earlier, is a variant of the multiple linear regression model. What sets the lasso regression apart is the way in which it punishes the inputs that contribute least to what we are trying to predict. In essence this method helps with both variable selection in this way by lowering coefficients of what is least associated with our output while creating predictions. In the end, the least associated variables with our outcome can have coefficients of 0, in essence removing them from our final model, making it simpler and more interpretable. In our model, changing the alpha changes the degree to which the coefficients of our inputs are punished if they are least associated with our outcome of interest.

The MLP Regressor is a variant of the deep neural network. An MLP Regressor is essentially a set of linear models that turn into a nonlinear model. It can be thought of almost like adding up a set of separate linear models into a nonlinear one. For each of these linear models within one layer, we will use some sort of error function like the sigmoid function to see how well each linear model predicts the output of interest. This can be repeated for as many hidden layers with as many nodes in each layer as desired. Once all these layers are connected, we can run our data through all the layers to a final output layer. In the output layer is where we achieve the combined nonlinear model with some sort of final scoring function (such as softmax). Based on these predictions, the error is distributed back over all of the nodes and the process is repeated many times, where the weights of each node within each layer change until training is complete. In our model we change the number of these hidden layers as well as the solver, which works to optimize the training of our models.

In the end, the SVR has the best MAE of our chosen candidates after cross validating amongst the potential hyperparameters. In order to try to optimize further, we can change the hyperparameters of our model further and compare results. The table below shows the changes to our grid search output on unseen data and resulting model MAE's in an attempt to further refine our winner's parameters

Change to hyperparameter / test size	**MAE**
No change (c=1, epsilon = 1, degree=3, gamma='auto', kernel = 'rbf'), test size = 30%	0.528
C=10	0.538
C = 100	0.581
C = 100, gamma=1	0.481
C = 1000, gamma=1	0.483
C = 100, gamma=1, test size = 20%	0.465

After playing with the test size and manipulating our C in conjunction with our gamma parameter, it looks like we can get some improvement by using 20% test data which helps our score as well! We are improved over our benchmark model in this category. On unseen data the MAE proves to be an improvement, now of 0.465. The tolerance accuracy (threshold = 1.0) is now observed at 86.3%. Further exploration of the possible hyperparameters of our model shows similar results, so this current model will be the comparison against our benchmark.

Results

Model Evaluation, Validation & Justification

We now have our final model and have compared it to a benchmark model in terms of overall absolute mean error. We have proven that we have made improvements to the model and obtained better results. However, we can directly compare a couple things to search for further improvement. Let's not only compare the two in terms of their overall tolerance accuracy, but peel the hood back further by seeing how often specific scores are correctly assessed.

We will first look at the tolerance accuracy for different thresholds on our baseline model and optimal model. After this, we will compare both directly using a matrix to see the comparative performance of different quality scores. The table below shows the improvements over a number of different threshold accuracies as well as the MAE.

Finally we can evaluate our scores over every possible wine quality score. As will be shown in the free form visualization, our optimal model has superior performance which will be discussed in more detail shortly.

Interestingly enough, it looks like we are getting a model that takes care of less frequent scenarios better in our optimal model. Specifically we can look at the quality score of 4 for all thresholds to see improvement and we can look at the quality of 7 for higher thresholds (which may also help explain the overall lowered MAE). Better performance for a score of 8 can be seen on the optimized model as well.

Overall, I point to the better MAE and better generalization to different quality scores of our optimal model as indicators of a stronger model versus a benchmark. Its overall threshold accuracies are superior as well.

Figures 8&9: Benchmark and Optimal Model Comparisons, Comparisons over all scores

```
benchmark set
            0.25      0.50      0.75       1.0
y_test
3       0.000000  0.000000  0.000000  0.000000
4       0.000000  0.020408  0.061224  0.122449
5       0.296375  0.462687  0.701493  0.855011
6       0.525292  0.770428  0.901427  0.980545
7       0.256705  0.352490  0.490421  0.651341
8       0.000000  0.017241  0.068966  0.120690
9       0.000000  0.000000  0.000000  0.000000
optimized set
            0.25      0.50      0.75       1.0
y_test
3       0.000000  0.000000  0.000000  0.000000
4       0.133333  0.133333  0.133333  0.200000
5       0.489051  0.667883  0.839416  0.934307
6       0.527897  0.748927  0.875536  0.929185
7       0.481928  0.596386  0.716867  0.813253
8       0.256410  0.307692  0.333333  0.410256
9       0.000000  0.000000  0.000000  0.000000
```

Variable	Benchmark	Optimal
Acc (T=0.25)	0.3778602350030921	0.48367346938775513
Acc (T=0.50)	0.559678416821274	0.6602040816326531
Acc (T=0.75)	0.7167594310451453	0.789795918367347
Acc (T=1.0)	0.8286951144094001	0.863265306122449
MAE	0.5452325792016529	0.4654686982340342

One more thing we need to think about is the robustness of our model. How is our model going to deal with varying datasets in terms of the data it is trained on and the data it predicts on? One way to accomplish this could be through Kfold cross validation: essentially we will cut the data into train and test sets with k folds. Each fold run can be looked into further to observe if our scores change drastically in terms of our MAE based on separate data cuts. The MAE of our model under these folds has risen slightly using a 3 fold cross validation with the Kfold method giving us an average of 0.674 (0.733, 0.665, and 0.623). I would argue that this is not too far off from where we landed with our results on unseen data, but points to the fact that directionally we are getting greater error when cross validating and shows some variation in our error.

Conclusion

Free-Form Visualization

Looking at our output accuracies within different thresholds in a visual format helps outline the importance of overfitting. Our general model is very good at predicting a score of 6, the most common quality score a critic has given.

Our optimal model shows that by being creative, using feature selection, data preprocessing, and testing multiple models we can get to an improved and informed solution that will generalize better to unseen data. This is important specifically in our problem: if one were to consistently give each wine a 6 that would be like saying every wine is "average" in its quality. This may work at an aggregated population level, but when making individual predictions this approach may fall short.

Figure 10: Unoptimized threshold accuracies

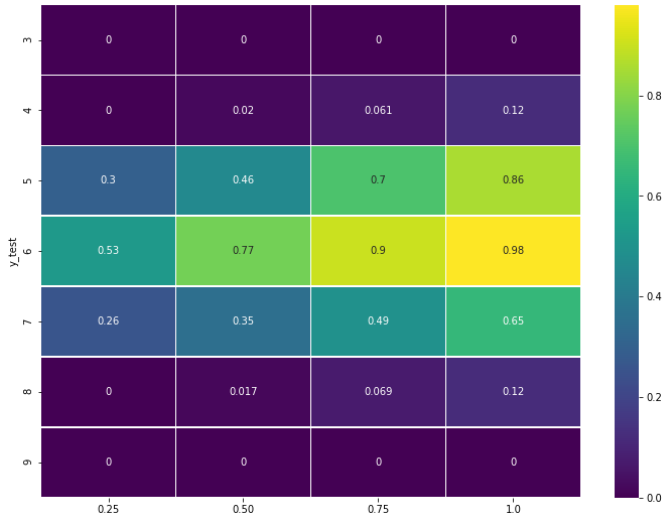
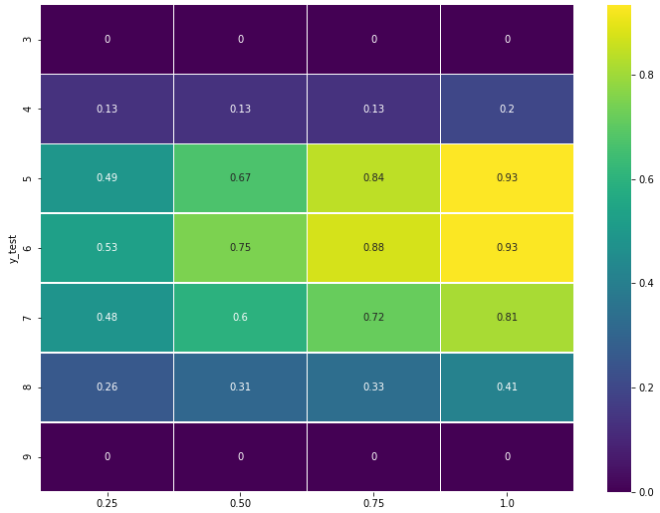


Figure 11: Optimized threshold accuracies



Reflection and Improvement

Accurate prediction of wine quality is not just a novel problem. There are direct relationships between the ability to lower the time and labor cost of paid professionals and consumer decision making when they choose a product.

In order to solve this problem, our approach began with exploratory analysis into the chemical data available to us. Given the nature of the data, it is hard to interpret the relationships between the molecular attributes of a given wine and what its quality is. What would even the most devoted wine lover tell you if you said the pH of a wine was 5.5 for instance? By looking at the variance within the different elements using a PCA, looking for multicollinearity amongst our inputs, and also by using a lasso regression to find the highest associations with our desired output of wine quality this problem is alleviated.

After understanding a bit more about our data, we moved forward to try and capture more intrinsic relationships about our data using a K means clustering methodology. We also removed variables that we empirically found unhelpful to solve the collinearity issue and lessen the complexity of our inputs which would help to inhibit overfitting.

With our final dataset in hand, we tested multiple potential models, each tuned amongst their hyperparameters. In reality we actually tested hundreds of models which is almost mind boggling to think about. Using a pipeline automated this process and we found the support vector regressor as the winner.

To validate the model, we tested the best model on a test set of unseen data and compared to our benchmark. We were successful in improving multi-level tolerance accuracy, lower our mean absolute error, and create better predictions for quality results that were further away from the mean.

The fact that our benchmark model is actually the untuned version of our best performing model made the task more difficult. The grid search results show, for example, that if we were to choose another model we could easily create a lot of improvement versus a poorer performing start. This made the tuning of the model and tweaking of the amount of data the model was trained on vital to success. The fact that there are very few very

high or low wine scores available also made the task more challenging when thinking about how to identify individual outlier scores accurately.

In terms of productionalization of our model, I would feel comfortable using the model in real life settings given one caveat. I feel that the amount of data we have trained our model on is pretty low compared to what is optimal. If we could have many more rows with our same inputs, the dates of the quality scores, and the individuals who gave these scores we could potentially create tensors based on each scorer in a time series. I have been doing research into implementing pytorch to run recurrent neural networks in these scenarios that show some robust results and slick options for taking care of missing data (for instance, if one quality scorer has 10 scores and everyone else has less, we could fill in the missingness for other individuals and even use a batching format to lower the missingness of our data).

If we could use these temporal relationships I think it would be really interesting to see what sort of accuracy and error we could produce. I feel like this could be an even more optimal solution to the problem.

Figure 12: Recurrent Neural Network Architecture Example: https://www.slideshare.net/xavigiro/recurrent-neural-networks-1-d2l2-deep-learning-for-speech-and-language-upc-2017