Introduction:

For this exercise I experimented with different preprocessing methods and their effect on various ML models as well as with and without hyperparameter optimization. I went with numerical preprocessors because every categorical preprocessor I tried had extremely long runtimes that I would just cancel after 15+ minutes of running.

Dataset Description:

I used the wine quality dataset[1] for this and picked the white wine dataset because it has more entries than the red dataset. This data includes 11 features and a target value called "quality" that is between 0 and 10.

[1]: https://archive-beta.ics.uci.edu/dataset/186/wine+quality

Experimental Setup:

For these experiments I picked a few models to test on, Random Forest because it has been the most accurate in every test on the wine dataset, K Nearest Neighbors because I wanted to how much preprocessing would improve it, and Support Vector Machine because I wanted a third model, and it was used in all the examples on scikit-learn's documentation.

Then I picked a couple preprocessing methods, I decided to use two different scaler methods because I thought those would have the most impact and I used the basic scaler and a minmax scaler.

Then I ran some tests on all the models with default parameters and optimized parameters with and without preprocessing and compared the results.

Parameters of the preprocessing methods are all default.

The hyperparameter optimizations will both use cross validation with 5 folds and have n_iter set to 15 which is the number of hyperparameter settings sampled.

The hyperparameters optimized are:

Random Forest:

bootstrap:

The default value for this is true, and I also tested false, this parameter either uses bootstrap samples for true or uses the whole dataset for each tree for false.

max_depth:

The default value for this is None, and I tested this with values 25, 50 and None, this parameter sets the max depth of the tree.

n_estimators:

The default value for this is 100, and I tested this with values 25, 50, 75, 100, 125 and 150, this parameter sets how many random trees the algorithm will generate.

min_samples_split:

The default value for this is 2, and I tested this with values 2, 4, 6, 8, this parameter determines the minimum amount of samples before a node can split.

min_samples_leaf:

The default value for this is 1, and I tested this with 1, 3, 5, 7, this parameter determines the minimum number of samples in a leaf.

K-Nearest-Neighbors:

n_neighbors:

The default value for this is 5, and I tested this with values 3, 4, 5, 6, 10, this parameter sets the number of neighbors.

leaf_size:

The default value for this is 30, and I tested this with values 10, 30, 50, 100, 200, this parameter sets the leaf size passed into the balltree and kdtree algorithms.

algorithm:

The default value for this is 'auto' and I tested this with 'ball_tree', 'kd_tree', 'brute', 'auto', this parameter changes which algorithm is being used.

p:

The default value for this is 2, and I tested this with both 1 and 2, it changes which distance formula is used.

weights:

The default value for this is 'uniform' and I also tested 'distance', uniform weighs all points in each 'neighborhood' equally, whereas distance weights points by the inverse of their distance.

Support Vector Machine:

C:

The default value for this is 1, I tested this with values 1, 10, 100, this parameter changes the regularization parameter.

kernel:

The default value for this is 'rbf' and I also tried 'sigmoid', I only went with these two because the other ones would never finish even when I left it running by itself for about 15 minutes, and they change how boundaries are calculated for classification.

NOTE: I could not get the parameter grid input on BayesSearchCV to work with preprocessors, so the results will reflect that by having all of the combinations of parameters for each preprocessing method, so the resulting graphs will be split by algorithm.

Standard Scaler[2]:

with_std:

Default value is true, this scales the data to 'unit variance', also tested with false.

with_mean:

Default value is true, this centers the data before scaling, also tested with false.

[2] https://scikit-learn.org/0.17/modules/generated/sklearn.preprocessing.StandardScaler.html

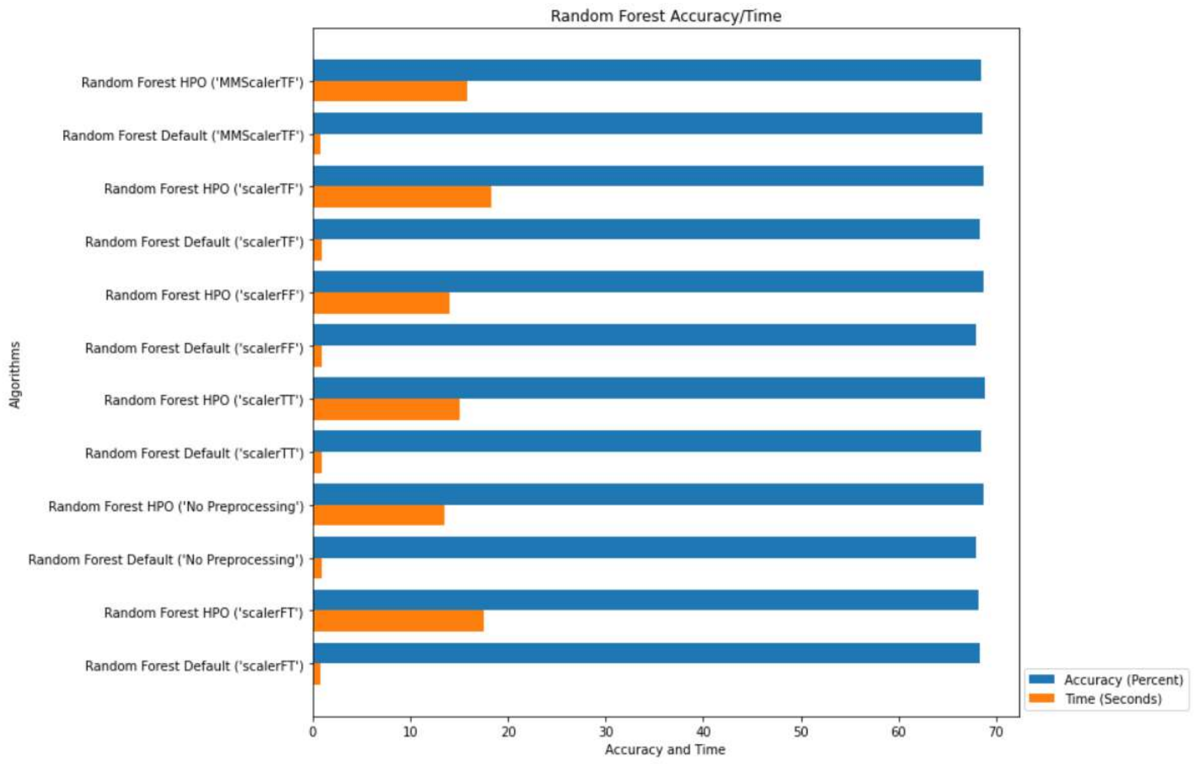
MinMax Scaler[3]:

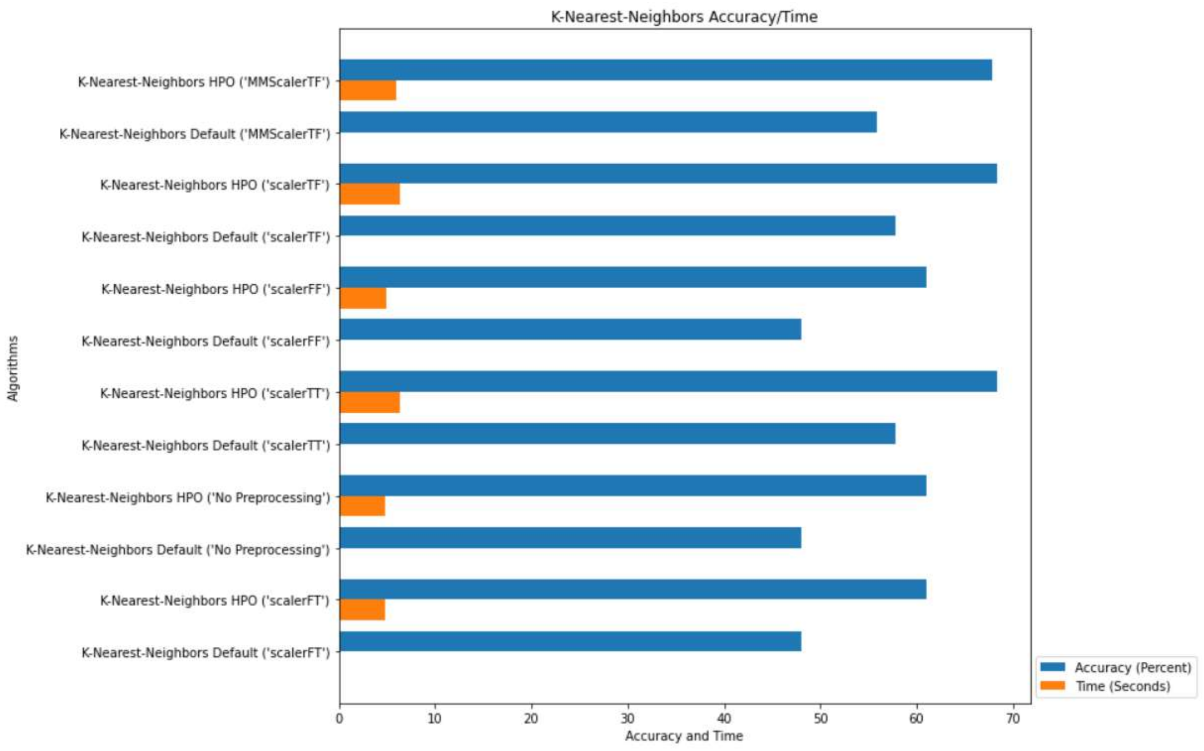
None of the parameters for this actually does anything for this problem, 'clip' when changed to true clips transformed values out of held out data to whatever feature range is inputted, 'copy' when set to false would do the normalizing inplace, and the feature_range was something I didn't get around to messing with but it lets me set the 'desired range of transformed data'.

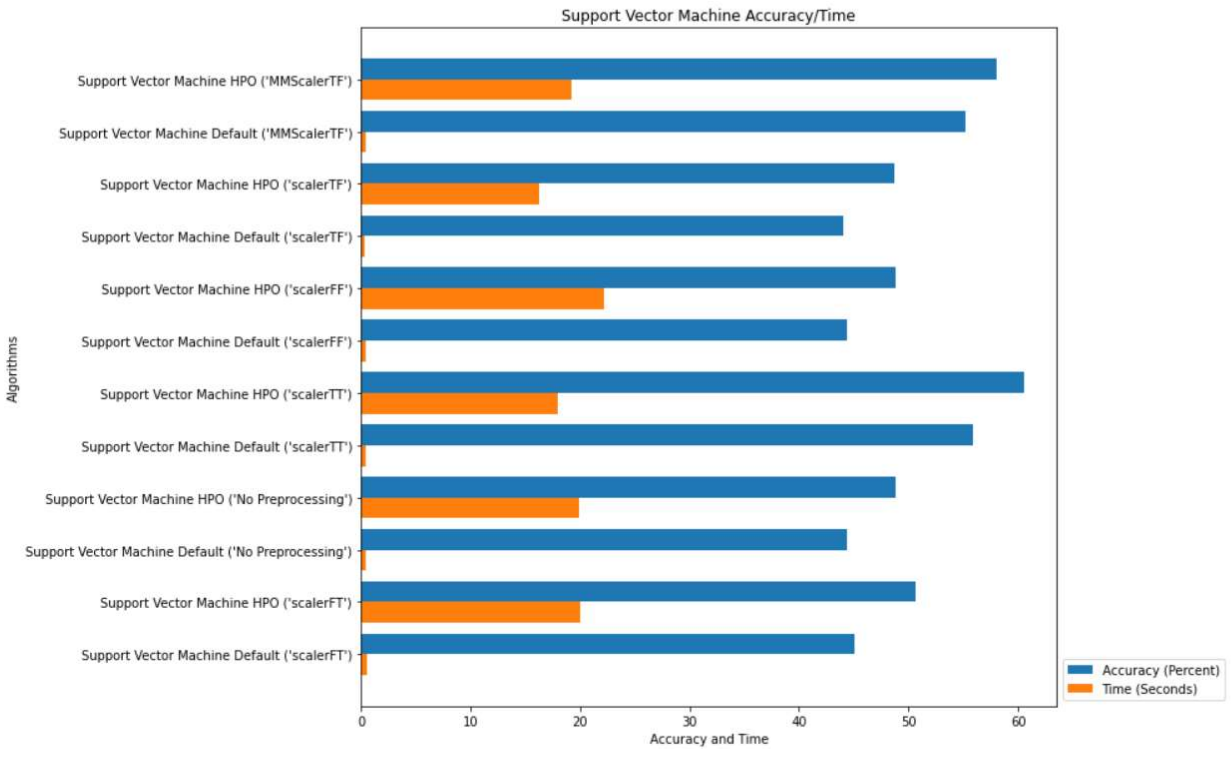
[3] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

I then output the results sorting by accuracy first and time second then labeling the best one.

Results:







```
Total time for models: 254.49128818511963
Best Algorithm by accuracy and time is: Random Forest HPO ('scalerTT')

List of algorithms sorted best to worst:
Random Forest HPO ('scalerTT'): 68.87755% accuracy, 15.07431 seconds
Random Forest HPO ('No Preprocessing'): 68.67347% accuracy, 13.45434 seconds
Random Forest HPO ('scalerFF'): 68.67347% accuracy, 14.06663 seconds
Random Forest HPO ('scalerTF'): 68.67347% accuracy, 18.35459 seconds
Random Forest Default ('MMScalerTF'): 68.57143% accuracy, 0.86374 seconds
Random Forest Default ('scalerTT'): 68.46939% accuracy, 0.87525 seconds
Random Forest HPO ('MMScalerTF'): 68.46939% accuracy, 15.85202 seconds
K-Nearest-Neighbors HPO ('scalerTF'): 68.36735% accuracy, 6.35856 seconds
K-Nearest-Neighbors HPO ('scalerTT'): 68.36735% accuracy, 6.37356 seconds
Random Forest Default ('scalerFT'): 68.26531% accuracy, 0.85974 seconds
Random Forest Default ('scalerTF'): 68.26531% accuracy, 0.87925 seconds
Random Forest HPO ('scalerFT'): 68.16327% accuracy, 17.53011 seconds
Random Forest Default ('No Preprocessing'): 67.95918% accuracy, 0.86574 seconds
Random Forest Default ('scalerFF'): 67.95918% accuracy, 0.87025 seconds
K-Nearest-Neighbors HPO ('MMScalerTF'): 67.85714% accuracy, 5.98463 seconds
K-Nearest-Neighbors HPO ('No Preprocessing'): 61.02041% accuracy, 4.75763 seconds
K-Nearest-Neighbors HPO ('scalerFT'): 61.02041% accuracy, 4.83844 seconds
K-Nearest-Neighbors HPO ('scalerFF'): 61.02041% accuracy, 4.93310 seconds
Support Vector Machine HPO ('scalerTT'): 60.51020% accuracy, 17.92614 seconds
Support Vector Machine HPO ('MMScalerTF'): 58.06122% accuracy, 19.25048 seconds
K-Nearest-Neighbors Default ('scalerTT'): 57.75510% accuracy, 0.01051 seconds
K-Nearest-Neighbors Default ('scalerTF'): 57.75510% accuracy, 0.01051 seconds
K-Nearest-Neighbors Default ('MMScalerTF'): 55.91837% accuracy, 0.01051 seconds
Support Vector Machine Default ('scalerTT'): 55.91837% accuracy, 0.38933 seconds
Support Vector Machine Default ('MMScalerTF'): 55.20408% accuracy, 0.38433 seconds
Support Vector Machine HPO ('scalerFT'): 50.61224% accuracy, 20.01113 seconds
Support Vector Machine HPO ('No Preprocessing'): 48.77551% accuracy, 19.93829 seconds
Support Vector Machine HPO ('scalerFF'): 48.77551% accuracy, 22.15964 seconds
Support Vector Machine HPO ('scalerTF'): 48.67347% accuracy, 16.28744 seconds
K-Nearest-Neighbors Default ('No Preprocessing'): 48.06122% accuracy, 0.00851 seconds
K-Nearest-Neighbors Default ('scalerFT'): 48.06122% accuracy, 0.01051 seconds
K-Nearest-Neighbors Default ('scalerFF'): 48.06122% accuracy, 0.01051 seconds
Support Vector Machine Default ('scalerFT'): 45.10204% accuracy, 0.49192 seconds
Support Vector Machine Default ('scalerFF'): 44.38776% accuracy, 0.40385 seconds
Support Vector Machine Default ('No Preprocessing'): 44.38776% accuracy, 0.44038 seconds
Support Vector Machine Default ('scalerTF'): 44.08163% accuracy, 0.36631 seconds


List of best hyperparameters per algorithm:
Best Paremeters found for Random Forest ('scalerFT', StandardScaler(with_std=False)): OrderedDict([('classifier__bootst
rap', False), ('classifier__max_depth', 25), ('classifier__min_samples_leaf', 1), ('classifier__min_samples_split', 4), ('classifier
__n_estimators', 125)])

Best Paremeters found for Random Forest ('No Preprocessing', None): OrderedDict([('classifier__bootstrap', False), ('clas
sifier__max_depth', 50), ('classifier__min_samples_leaf', 1), ('classifier__min_samples_split', 8), ('classifier__n_estimators', 5
0)])

Best Paremeters found for Random Forest ('scalerTT', StandardScaler()): OrderedDict([('classifier__bootstrap', False), ('cl
assifier__max_depth', 25), ('classifier__min_samples_leaf', 3), ('classifier__min_samples_split', 4), ('classifier__n_estimators
', 100)])

Best Paremeters found for Random Forest ('scalerFF', StandardScaler(with_mean=False, with_std=False)): OrderedDict
([('classifier__bootstrap', False), ('classifier__max_depth', 50), ('classifier__min_samples_leaf', 1), ('classifier__min_samples
_split', 8), ('classifier__n_estimators', 50)])

Best Paremeters found for Random Forest ('scalerTF', StandardScaler(with_mean=False)): OrderedDict([('classifier__boo
tstrap', False), ('classifier__max_depth', 25), ('classifier__min_samples_leaf', 3), ('classifier__min_samples_split', 2), ('classif
ier__n_estimators', 150)])

Best Paremeters found for Random Forest ('MMScalerTF', MinMaxScaler()): OrderedDict([('classifier__bootstrap', False),
('classifier__max_depth', 25), ('classifier__min_samples_leaf', 3), ('classifier__min_samples_split', 6), ('classifier__n_estimat
ors', 125)])
```

```
Best Paremeters found for K-Nearest-Neighbors ('scalerFT', StandardScaler(with_std=False)): OrderedDict([('classifier__
algorithm', 'kd_tree'), ('classifier__leaf_size', 200), ('classifier__n_neighbors', 10), ('classifier__p', 1), ('classifier__weights', '
distance')])

Best Paremeters found for K-Nearest-Neighbors ('No Preprocessing', None): OrderedDict([('classifier__algorithm', 'kd_tre
e'), ('classifier__leaf_size', 200), ('classifier__n_neighbors', 10), ('classifier__p', 1), ('classifier__weights', 'distance')])

Best Paremeters found for K-Nearest-Neighbors ('scalerTT', StandardScaler()): OrderedDict([('classifier__algorithm', 'aut
o'), ('classifier__leaf_size', 200), ('classifier__n_neighbors', 10), ('classifier__p', 1), ('classifier__weights', 'distance')])

Best Paremeters found for K-Nearest-Neighbors ('scalerFF', StandardScaler(with_mean=False, with_std=False)): Ordere
dDict([('classifier__algorithm', 'kd_tree'), ('classifier__leaf_size', 200), ('classifier__n_neighbors', 10), ('classifier__p', 1), ('cl
assifier__weights', 'distance')])

Best Paremeters found for K-Nearest-Neighbors ('scalerTF', StandardScaler(with_mean=False)): OrderedDict([('classifie
r__algorithm', 'auto'), ('classifier__leaf_size', 200), ('classifier__n_neighbors', 10), ('classifier__p', 1), ('classifier__weights', '
distance')])

Best Paremeters found for K-Nearest-Neighbors ('MMScalerTF', MinMaxScaler()): OrderedDict([('classifier__algorithm', '
kd_tree'), ('classifier__leaf_size', 200), ('classifier__n_neighbors', 10), ('classifier__p', 1), ('classifier__weights', 'distance')])

Best Paremeters found for Support Vector Machine ('scalerFT', StandardScaler(with_std=False)): OrderedDict([('classifie
r__C', 100), ('classifier__kernel', 'rbf')])

Best Paremeters found for Support Vector Machine ('No Preprocessing', None): OrderedDict([('classifier__C', 100), ('classi
fier__kernel', 'rbf')])

Best Paremeters found for Support Vector Machine ('scalerTT', StandardScaler()): OrderedDict([('classifier__C', 100), ('cl
assifier__kernel', 'rbf')])

Best Paremeters found for Support Vector Machine ('scalerFF', StandardScaler(with_mean=False, with_std=False)): Orde
redDict([('classifier__C', 100), ('classifier__kernel', 'rbf')])

Best Paremeters found for Support Vector Machine ('scalerTF', StandardScaler(with_mean=False)): OrderedDict([('classi
fier__C', 100), ('classifier__kernel', 'rbf')])

Best Paremeters found for Support Vector Machine ('MMScalerTF', MinMaxScaler()): OrderedDict([('classifier__C', 100),
('classifier__kernel', 'rbf')])
```

The results show some improvements for random forest with preprocessed data and huge improvements for the support vector and k nearest neighbors algorithms when the data is preprocessed. Also that performance is improved across the board when the data is preprocessed for this dataset.

Output for best parameters includes the parameters for the preprocessor, if the value was default it didn't show up so I have named them by which values are true and false starting with 'with_mean' then 'with_std'.

I would have liked to pass their parameters into the optimizer with all the possible values instead of hardcoding them, but I couldn't figure out how to do it.

Code:

Main.py