CS 598 PSL

Project 1

xxxx (email)

Objective

The goals of this project were threefold:

1. Read in and preprocess training data from the Ames housing dataset
2. Fit several models on the preprocessed training data
3. Read in and preprocess test data, then use the fitted models to predict sale prices

Three different types of models were offered as choices for #2 above: linear regression models, tree models, and generalized additive models. For the purposes of this project, the two models chosen were a linear model with Elastic Net penalty, and a tree-based gradient boosting model.

Implementation Details

Preprocessing

xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.

1. Fix known bad data

As discussed on Piazza (@224 (https://piazza.com/class/kdf6l5f8bb78j?cid=224)), there are some missing and bad values for the `Garage_Yr_Blt` variable. Missing values were be replaced with `0` s, and the one obviously incorrect value `2207` was changed to 2007.

2. Winsorize/clip variables

xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.
xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.

xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.

> Bsmt_Unf_SF, BsmtFin_SF_2, Enclosed_Porch, First_Flr_SF, Garage_Area, Gr_Liv_Area, Lot_Area, Lot_Frontage, Mas_Vnr_Area, Misc_Val, Open_Porch_SF, Screen_Porch, Second_Flr_SF, Three_season_porch, Total_Bsmt_SF, Wood_Deck_SF

3. Apply categorical datatypes

xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.

xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.

xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx

4. Drop near-homogeneous variables

xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.

> Alley, Bsmt_Half_Bath, Central_Air, Condition_2, Electrical, Functional, Garage_Cond, Heating, Kitchen_AbvGr, Land_Slope, **Latitude**, **Longitude**, Low_Qual_Fin_SF, Misc_Feature, Misc_Val, Paved_Drive, Pool_Area, Pool_QC, Roof_Matl, Screen_Porch, Street, Three_season_porch, Utilities

5. Split sale price and remove identifier

xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.

6. Test data shape matched to training dataframe

xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.

Models

1. Linear with Elastic Net penalty

For the linear model, the `sklearn.linear_model.ElasticNetCV` estimator was used. The only relevant modeling parameters specified were:

parameter	value	note
l1_ratio1	[0.1, 0.3, 0.5, 0.6, 0.7, 0.8, 0.9, 0.95, 0.99, 1]	Arbitrarily chosen values, getting closer when approaching 1 (Ridge)
normalize	True	
cv	10	

The estimator defined its own `alphas` sequence, and chose an α value by cross-validation, which typically resulted in something on the order of `~1e-5` . Unlike the `glmnet` package in R which offers a choice between α_{min} and α_{1se}, the Python `sklearn` estimators do not (directly) provide this as a parameter, and always use α_{min}. Although it is possible to identify the α_{1se}, during this project α_{min} was used for convenience.

2. Tree-based gradient boosting

For the tree-based model, the `xgboost.XGBRegressor` estimator was used. Hyperparameter tuning was done in a separate file (not included), roughly following the procedure described [here (https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/)](https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/). Typically, hyperparameters would not be hardcoded, but in this case it was acceptable in order to reduce computation time when evaluating submitted code (see Piazza [@644 (https://piazza.com/class/kdf6l5f8bb78j?cid=644)](https://piazza.com/class/kdf6l5f8bb78j?cid=644)). Resulting parameters (tuned in **bold**):

parameter	value	note
objective	reg:squarederror	
eta	0.01	
n_estimators	3997	
max_depth	3	Seems shallow but produces adequate results
min_child_weight	6	
gamma	0.0	
subsample	1.0	
colsample_bytree	1.0	
reg_alpha	~0.063	
reg_lambda	~3.98	

Stuent should add a description on the meaning of those parameters

Performance

Runtime and accuracy

Data was collected by running/timing the submitted `mymain.py` file and evaluating predictions from the resulting files; 10-split loop was done using a separate Jupyter notebook for convenience (not included). Splits were specified using the provided `project1_testIDs.dat` file. For both estimators, the `n_job` parameter was set to -1 to use all available CPU cores. Accuracy table and timings below:

split#	time	linear RMSE	tree RMSE
0	18.27s	0.120	0.120

split#	time	linear RMSE	tree RMSE
1	14.88s	0.122	0.121
2	18.60s	0.114	0.114
3	17.97s	0.129	0.118
4	15.21s	0.115	0.111
5	16.67s	0.130	0.127
6	18.65s	0.131	0.130
7	14.72s	0.124	0.122
8	17.64s	0.130	0.127
9	16.43s	0.123	0.118

Platform

All code was run on a Windows 7 64-bit PC, with an Intel i5-3570K @3.4GHz, 8192MB RAM, and a GPU which did not support XGBoost GPU algorithms. Python module versions were: Python 3.8.6, jupyterlab 2.2.6, numpy 1.17.3, pandas 1.1.2, sklearn 0.0, and xgboost 1.2.0.

Conclusion

A minimal amount of preprocessing is *required* for both models due to their incompatibility with certain data, specifically missing data (`NaN`) and nominal variables. Even starting out by *just dropping nominal variables* and correcting missing data (i.e. no targeted preprocessing), both models (untuned) were already producing reasonably approximate predictions (`(U)` columns below). More intentional and thorough preprocessing (`(P)` columns below) based on observations of the data seems to have a much bigger impact on the performance of the linear model. The untuned tree-based model still sees some benefit from said preprocessing, but less so. Finally, hyperparameter tuning for `XGBoost` also produced some gains (`(H)` column below), but the actual tuning took some time to carry out. Thorough tuning may not always be worth the time investment.

split #	Linear (U)	Tree (U)	Linear (P)	Tree (P)	Linear Δ	Tree Δ	Tree (H)	Tree (H) Δ
0	0.207	0.156	0.120	0.126	-42%	-19%	0.120	-5%
1	0.179	0.158	0.122	0.134	-32%	-15%	0.121	-10%
2	0.171	0.152	0.114	0.130	-33%	-14%	0.114	-12%
3	0.202	0.152	0.129	0.132	-36%	-13%	0.118	-11%
4	0.162	0.142	0.115	0.119	-29%	-16%	0.111	-7%
5	0.215	0.170	0.130	0.126	-40%	-26%	0.127	1%
6	0.192	0.169	0.131	0.145	-32%	-14%	0.130	-10%
7	0.183	0.155	0.124	0.128	-32%	-17%	0.122	-5%
8	0.211	0.160	0.130	0.135	-38%	-16%	0.127	-6%
9	0.175	0.157	0.123	0.129	-30%	-18%	0.118	-9%

It seems most important to focus efforts on exploring data and targeting easy preprocessing steps. That said, establishing an automated hyperparameter tuning procedure would enable unattended tuning for `XGBoost` . In fact, one interesting observation is that *almost all* of the preprocessing steps taken for this project seems like they could've been done without supervision; it doesn't require human judgement to identify variables that are nearly-homogeneous (for drop), skewed right (for clipping), or to apply well-defined discrete levels to ordinal variables. It might be an interesting exercise to repeat this project without directly specifying the individual preprocessing steps, though one immediate problem with that would be the extraction of the discrete levels for ordinal variables from source materials.