Capstone Project

Machine Learning Engineer Nanodegree

Celso Araujo

May 31th, 2017

I. Definition

Project Overview

How could someone determine the price of a house? This house is for sale: is it cheap, or is the seller asking an exorbitant price for it? What is a “fair” price for a house containing 3 bedrooms, a garage for 2 cars and located in a certain vibrant neighborhood downtown?

The questions above can be somewhat quite difficult to answer. While it can be quite obvious that, eg, big houses are more expensive than small houses, or that houses in neighborhood X are cheaper than those on neighborhood Y, it would be nice to see if these assumptions hold, and translate them into numbers.

In this project, I built some house price prediction models based on house prices from Ames, Iowa, USA. Inspiration and data comes from a Kaggle[1] competition, and the models will be tested against the competition's data.

Problem Statement

We are provided with the "Ames Housing Dataset", which contains a number of houses, each house being described by a number of features (size, location, number of bedrooms etc). We know the sale price of roughly half of the houses (training set); the goal is to accurately predict the sale prices for the other half (test set).

Data preprocessing may include handling missing values, codifying categorical variables as numeric variables, and feature engineering (variable transformations, summing/combining and discarding variables etc). Different regression methods will be used to tackle the problem, like linear regression and gradient boosting trees. Model parameters will be fine tuned with help of k-fold cross validation and grid search. In the end, the best methods will be compared against each other and a new model being a combination of the best models will be tried and tested.

Metrics

Kaggle's competition uses the RMSLE (root mean squared logarithmic error) as metric. I will use the same metric in the project.

What is RMSLE? It is a variation of the commonly-used RMSE measure for regression error. If n is the sample size, p_i is the predicted value for the i-th item and a_i is its the actual value, then RMSE is:

$$RMSE=\sqrt{\frac{1}{n}\sum_{i=1}^{n}(p_i-a_i)^2}$$

1 https://www.kaggle.com/c/house-prices-advanced-regression-techniques/

The closer RMSE is to zero, the better our predictions are: *RMSE = 0* means all predictions are correct.

RMSLE is the RMSE calculated over the logarithm of the predicted and actual values (in fact, we will use *log(value + 1)* to avoid calculating the logarithm of zero). If *log(x)* is the natural logarithm of *x*, we thus have:

$$RMSLE=\sqrt{\frac{1}{n}\sum_{i=1}^{n}(\log(p_i+1)-\log(a_i+1))^2}$$

RMSLE is chosen because it only accounts to the percentual differences between predicted and actual values, not the absolute differences, as is with RMSE. So, in a way, RMSLE averages the square of the percentual differences between predicted and actual values.

There is a very interesting relationship between RMSE and RMSLE which will be very helpful later on. Suppose we apply a log transform to our target variable (price). Let's call our new variable *z*:

$$z=\log(p+1),$$

so that, after we predict *z*, we apply an inverse transform to predict the actual price:

$$p=e^z-1.$$

If we apply the same transform to the actual price letting *w = log(a + 1)* and look at both the RMSE and RMSLE formulas, we conclude that the RMSE of the log-transformed target variable is exactly the RMSLE of the original target variable! So, we don't really need to apply the inverse transform and calculate the RMSLE in order to calculate our score.

II. Analysis

Data Exploration

URL for the dataset: https://www.kaggle.com/c/house-prices-advanced-regression-techniques/data

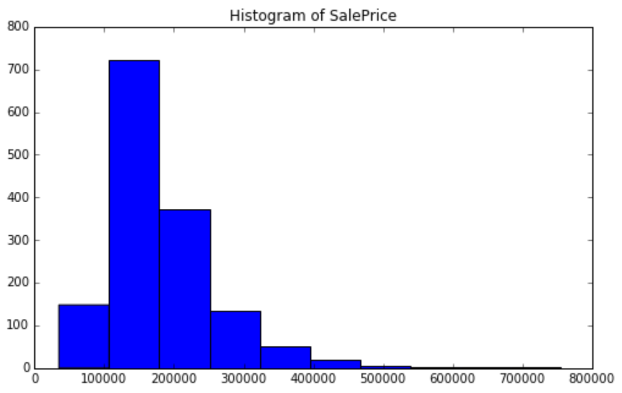
Data size:

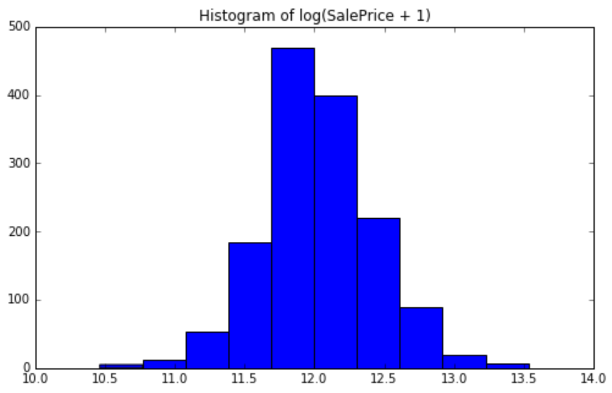
- 2919 instances (houses)
- 79 features (input variables)
- 1 target variable (sale price, real-valued)
  - 1460 houses have their sale price listed (training set)
  - 1459 houses are not provided with their sale prices (test set)

Features:

- 35 numerical features (not counting the target variable here). They include date-related features. Examples: lot size, living area above ground, number of bathrooms, year sold, year garage was built.
  - Some numerical features may be considered ordinal. For example, *OveralCond* is an integer between 1 to 10, 1 meaning “very poor” and 10 meaning “very excellent”.
- 44 categorical features
  - About 15 non-binary features may be considered ordinal. Examples: kitchen quality, slope of property (gentle/moderate/severe), interior finish of garage
  - Other categorical features. Examples: electrical system, type of foundation, neighborhood (location)

The target variable, SalePrice, on the training set, is quite skewed. However, if we apply a log transform on it (actualy, log(SalePrice + 1), its distribution becomes more similar to a normal curve. Minimum SalePrice is *34,900*, maximum is *755,000*, the mean value is *180,921* and the median is *163,000*.





There are missing values in some features, both on training and test sets. For some categorical variables, a missing value actually means something, eg, the house lacks some feature. But, for numerical variables, those missing values will need some kind of treatment. This will be discussed later. Here is a table presenting the missing values count:

Variable	Missing count		Type	Variable	Missing count		Type
	Training	Test			Training	Test	
Alley	1369	1352	Categorical	GarageArea		1	Numerical
BsmtCond	37	45	Categorical	GarageCars		1	Numerical
BsmtExposure	38	44	Categorical	GarageCond	81	78	Categorical
BsmtFinSF1		1	Numerical	GarageFinish	81	78	Categorical
BsmtFinSF2		1	Numerical	GarageQual	81	78	Categorical
BsmtFinType1	37	42	Categorical	GarageType	81	76	Categorical
BsmtFinType2	38	42	Categorical	GarageYrBlt	81	78	Numerical
BsmtFullBath		2	Numerical	KitchenQual		1	Categorical
BsmtHalfBath		2	Numerical	LotFrontage	259	227	Numerical
BsmtQual	37	44	Categorical	MasVnrArea	8	15	Numerical
BsmtUnfSF		1	Numerical	MasVnrType	8	16	Categorical
Electrical	1		Categorical	MiscFeature	1406	1408	Categorical
Exterior1st		1	Numerical	MSZoning		4	Categorical
Exterior2nd		1	Numerical	PoolQC	1453	1456	Categorical
Fence	1179	1169	Categorical	SaleType		1	Categorical
FireplaceQu	690	730	Categorical	TotalBsmtSF		1	Numerical
Functional		2	Categorical	Utilities		2	Categorical

The only numerical variables with missing values on the training set are *LotFrontage* and *MasVnrArea*. On this set, all other variables with missing values are categorical. The test set has more variables with missing values overall.

Algorithms and Techniques

All computing was done and implemented on Python 2.7, with extensive use of scikit-learn[2], Pandas[3] and NumPy[4] libraries.

2 http://scikit-learn.org/
3 http://pandas.pydata.org/
4 http://www.numpy.org/

After some feature engineering on both training and test sets, which will be discussed later, four regression algorithms available in scikit-learn library will be separately applied to the training set: Ridge Regression[5], Gradient Boosting Regressor[6], Random Forest Regressor[7] and Extra Trees Regressor[8].

Training data will be partitioned for cross validation in 5 folds. For each regression algorithm, a randomized grid search will be ran over the cross validation training set in order to adjust the algorithm parameters and find the best model. The best model will be the one with the best validation set score, measured in terms of the RMSLE.

Each algorithm, with its best model, will be fitted in the test dataset and the resulting predictions will be submitted to Kaggle. The results will be compared both to our benchmark and the competition's leaderboard.

In the end, the models will be combined: a weighted average of their predictions will be considered as a new prediction, and this will be a new model. This new model will be tested on the 5 folds of the training set and its validation score will be recorded. Results will be sent to Kaggle for final validation.

A word about the proposed algorithms

The first algorithm, Ridge Regression, is a variation of the ordinary least-squares regression. In ordinary least-squares, for a known matrix *A* and a known vector *B*, we try to find *x* (vector containing feature weights) that minimizes $||Ax-b||^2$, whereas in Ridge Regression, for a given, nonnegative parameter *α*, we try instead to minimize $||Ax-b||^2+||\alpha Ix||^2$, where *I* is the identity matrix, in order to control de variance of the output.

Random Forest and Extra Trees are two related regression algorithms which randomly build decision trees. A number of trees are built (*n_estimators* parameter) and the trees are built from samples from the training set, with or without replacement (*bootstrap* parameter). The minimum number of elements in a given node required to split this node is given by the *min_samples_split* parameter. The final prediction is given by an average of the individual trees' predictions. The main difference between Random Forest and Extra Trees is that Extra Trees uses a random threshold to split a node and Random Forest doesn't. They are very parallelizable algorithms.

Gradient Boosting Trees tries to fit a number of regression trees to the (negative) gradient of a given loss function (*loss* parameter). Here, the parameter *max_depth* controls, as suggested, the maximum depth of the trees. This algorithm usually produces better results than Random Forest or Extra Trees, at the cost of not being parallelizable.

Benchmark

For each calculated final model, sale prices will be predicted from the test set and the results will be submited to Kaggle, which will compare against the (unknown to me) real sale prices and calculate its RMSLE score.

Kaggle provides a sample submission file for the competition. I submited it with no modifications, and it scored *0.40890*. Originally, my first aim was to do better than that.

A new benchmark will be used. I created a very simple model predicting the house price to be the average house price for its neighborhood. After submiting to Kaggle, it scored *0.27484*, which is much better than the original benchmark. So, it will be used instead as the benchmark for the upcoming models.

5 http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html
6 http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html
7 http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html
8 http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesRegressor.html

At the time of submission, it scored the 2137th position out of 2344 on the leaderboard, staying in the 91th percentile.

III. Methodology

Data Preprocessing

Training and test data, as noted earlier, contain missing values and some categorical variables. Here, data was preprocessed to handle those two issues.

No special treatment was done to ordinal variables, being them numerical or categorical.

Missing values were treated on the same way on the training and the test sets. No variable or data was removed from any set. All missing values received an actual value. For each set:

- Numerical variables: each missing value received the median value of this variable over its Neighborhood.
- Categorical variables: the missing values simply received the value 'N/A'.

Then, categorical variables were transformed into numerical variables using a one-hot encoding[9] technique: if variable A has $A_1, ..., A_n$ as possible values, then we create n binary variables named $A.A_1, ..., A.A_n$, where one of them will be valued as 1, according to A's value, and the others will be zero-valued. The original, categorical, variable will be dropped out.

The target variable, *SalePrice*, will also be modified. The models will be trained both against the original variable and the transformed one, which will be called *logSalePrice*:

$$logSalePrice = \log(SalePrice + 1)$$

As noted earlier, this log-transform removed most of the skewness of the original variable, and made it more similar to a normal-distributed variable.

No regard was given to possible outliers on the dataset. That is, outliers were not looked for, removed or modified.

Implementation

After processing data as just described, four regression algorithms available in scikit-learn library will be separately applied to the training set.

Training data will, at first, be randomly partitioned in 5 folds[10] for cross validation purposes. Then, for each algorithm, the following will be done:

- Fit 100 models to the training data, using a randomized grid search[11] over a specified set of parameters in order to find the best model:
  - For each one of the 5 folds, retain 4 folds to use as a training set and use the remaining as a validation set. Fit the model to the training set and apply the resulting model to the validation set.
  - The score for this model will be the mean RMSLE over each validation set.
- The best model will be defined as the model with the best (minimum) mean validation RMSLE. This number will be used as the algorithm validation score.

The following algorithms will be applied, together with the following range of parameters for randomized grid searches:

9 https://pandas.pydata.org/pandas-docs/stable/generated/pandas.get_dummies.html
10 http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html
11 http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html

- Ridge Regression (Ridge):
  - alpha: real-valued, uniformly distributed between 0.3 and 5
- Gradient Boosting Regressor (GBR):
  - min_samples_split: integer, uniformly distributed between 2 and 20
  - n_estimators: integer, uniformly distributed between 50 and 5000
  - max_depth: integer, uniformly distributed in [3, 4, 5]
  - loss: string, uniformly distributed between 'ls' and 'lad'
- Random Forest Regressor (RF):
  - min_samples_split: integer, uniformly distributed between 2 and 20
  - n_estimators: integer, uniformly distributed between 50 and 5000
  - bootstrap: boolean, uniformly distributed between True and False
- Extra Trees Regressor (Extra):
  - min_samples_split: integer, uniformly distributed between 2 and 20
  - n_estimators: integer, uniformly distributed between 50 and 5000
  - bootstrap: boolean, uniformly distributed between True and False

Each algorithm, with its best model, will be fitted in the test dataset and the resulting predictions will be submitted to Kaggle. The results will be compared both to our benchmark and the competition's leaderboard.

In the end, the models will be combined: a weighted average of their predictions will be considered as a new prediction, and this will be a new model. This new model will be tested on the 5 folds of the training set and its validation score will be recorded. If a better result is achieved, it will be sent to Kaggle.

Model predictions will be averaged using their validation scores as weights, and it will be done in two ways: using directly the inverse scores as weights (remember: for our score, smaller is better), or using the inverse square the scores to set up the weights. So, in the end, we can have different combination models with the same model-specific parameters.

Model Combinations (Averaging)

Suppose I have fitted *k* models to our data and made predictions to the test set. Let's call $p_{i,j}$ the prediction for the i^{th} data point (our dataset containing *n* points) given by the j^{th} model. Let s_j be the calculated validation score (RMSLE) for model *j*. Then I will define the combined model to be a weighted average of the predictions of the considered models:

$$\bar{p}_i = \sum_{j=1}^{k} w_j \, p_{i,j}, \, for \; i = 1, \ldots, n \,,$$

where w_j is the (normalized) weight corresponding to the j^{th} model. Two weighting methods are proposed here. Let s_j be the score (RMSLE) for model *j*. Remember: our scores are nonnegative, and the smaller the better. The first weighting method, which I simply call “inverse scores”, sets the (normalized) weights to be the inverse of the scores:

$$\hat{w}_j = \frac{1}{s_j} for \; j = 1, \ldots, k \text{ , and then normalize: } w_j = \frac{\hat{w}_j}{\hat{w}_1 + \hat{w}_2 + \ldots + \hat{w}_k}$$

The second weighting method, which I call “inverse square”, sets the (normalized) weights to be the inverse of the squared scores:

$\hat{w}_j = \frac{1}{s_j^2}$ *for* $j=1,\dots,k$, and then normalize: $w_j = \frac{\hat{w}_j}{\hat{w}_1 + \hat{w}_2 + \dots + \hat{w}_k}$

IV. Results

Model Evaluation and Validation

Models were trained both against the original target variable *SalePrice* and against a log-transformed version of it, *log(SalePrice + 1)*. The table below shows the best model found on each case, together with its validation RMSLE.

Model	Parameters (original)	Parameters (log)	Validation RMSLE (original)	Validation RMSLE (log)
Ridge	alpha: 1.08854}	alpha: 0.53638	0.145312	0.144199
GBR	min_samples_split: 19 loss: 'lad' n_estimators: 3190 max_depth: 5	min_samples_split: 7 loss' 'lad' n_estimators: 2935 max_depth: 4	0.125915	0.123418
RF	min_samples_split: 4 n_estimators: 718 bootstrap: True	min_samples_split: 2 n_estimators: 2200 bootstrap: True	0.143981	0.142067
Extra	min_samples_split: 4 n_estimators: 3470 bootstrap: False	min_samples_split: 4 n_estimators: 718 bootstrap: True	0.146739	0.143506
Combinations	weights: inverse scores	weights: inverse scores	0.129392	0.126775
Combinations	weights: inverse square scores	weights: inverse square scores	0.128717	0.125776
Combinations minus GBR	weights: inverse square scores	weights: inverse square scores	0.135025	0.133096

Here, Combinations is a model with the best found Ridge, GBR, RF and Extra models combined, as described above.

After I got the first 6 models above, I noted that GBR performed better than any model, including the combined models. So, I decided to add a 7th model, a combined model without GBR.

We can see that every model had a better performance against the log-transformed *SalePrice* than against the original target variable. Also, for the combined models, the inverse squared weights worked better than just the inverse scores as weights. In the end, I decided to submit to Kaggle the log-transformed models, as they scored systematically than the models trained against the original target variable.

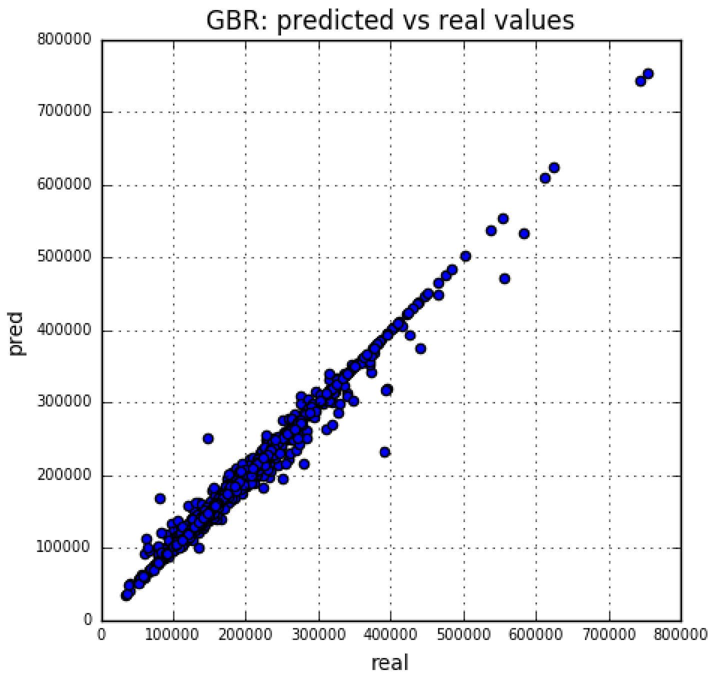
Model	Validation RMSLE (log)	Kaggle score
Ridge	0.144199	0.13619
GBR	0.123418	0.12913
RF	0.142067	0.14525
Extra	0.143506	0.14799
Combinations (inverse scores)	0.126775	0.13089
Combinations (inverse square scores)	0.125776	0.13050
Combinations minus GBR (inverse square scores)	0.133096	0.13582

The best model, GBR, granted me at the time the 1000th position on the leaderboard, out of 2248 positions, staying in the 45th percentile.

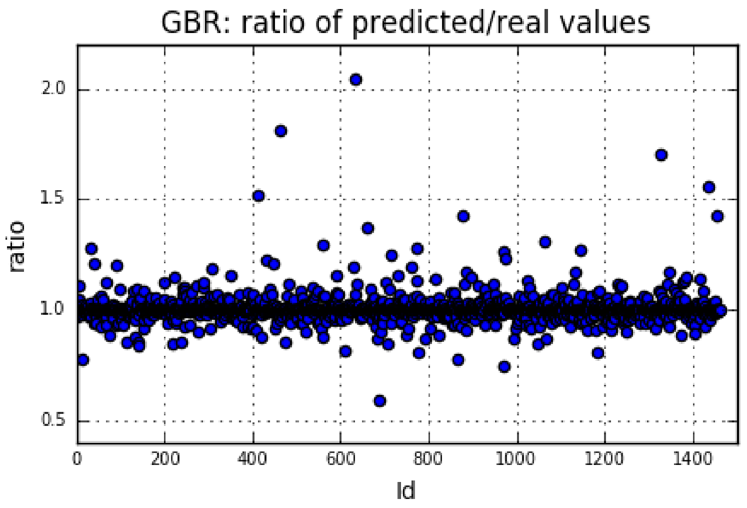
V. Conclusion

A Quick Analysis of the Best Model

Our score metric, RMSLE, surely is a useful way to measure a model performance. But it's not the only way to do it. The following graphic shows how the predicted values calculated by our best model relate to the actual values, when applying the model to our full training set:



We can see that, apparently, we have a good model, with a few bad predictions here and there. But here we are comparing absolute values: predicting a $50,000 deviation from the actual price may be no really big deal when we talk about a $700,000 house, but it is very significant when we talk about a $200,000 house. So, let's take the ratio between predicted and real values and see the relative difference between them, instead. The closer to 1.0, the better:



We can see that most of the predictions are accurate to within 20% (up or down). In fact, it's better than that: about 94% of our predictions are within 10% of the actual price, and 83% of them are within 5% of the actual price. However, we have a few very wrongly predicted prices, with more than 50% of deviation. They could be considered as outliers, and a closer look could be given into those houses.

Unfortunately, the actual prices from the test set are not available, so I cannot make this analysis for the test set, which would be ideal.

The following table shows the features which were considered the most important ones by our best model:

Feature	Importance
GrLivArea	6.18%
OpenPorchSF	5.73%
TotalBsmtSF	5.44%
2ndFlrSF	4.96%
MasVnrArea	4.93%
1stFlrSF	4.89%
EnclosedPorch	4.77%
WoodDeckSF	4.09%
LotArea	4.00%
GarageArea	3.66%
BsmtFinSF1	3.62%
LotFrontage	3.24%
BsmtUnfSF	2.77%
YearBuilt	2.14%
ScreenPorch	1.91%

We can conclude that, according to our model, many of the most important features when calculating the price of a house are related to the size of the house (ground living area, 1st and 2nd floors square feet, basement square feet, lot frontage, lot area, garage area), and it makes sense: people do pay more for bigger houses.

Reflection

To determine a "fair" price for a house is no easy task. The same should be said about predicting actual prices. Here, inspired by a Kaggle competition, a method to make house price predictions was proposed. Features were extracted and engineered, data was partitioned and several regression algorithms were applied with help of grid search to find the best parameters.

It was shown that it may be useful to create a new model by combining (averaging) predictions from other models. The "Combinations minus GBR" model, which is a weighted average of Ridge, RF and Extra models, performed better than any of its individual component models. However, it's interesting to note that GBR alone scored better than any model, including when combined to the other models.

All results were better than the proposed benchmarks, so I consider this job a success. However, the position attained on Kaggle's leaderboard suggests there is still a quite big room for improvement. Some possible and non-exclusive paths could be tried in the future to improve this task and get better results:

- Feature engineering: here, almost all of the original features were used, and all categorical features were converted to binary via one-hot encoding, adding a large bunch of new features to the model. No other kind of transform was applied to the independent variables. A more detailed look could be taken into the variables to see whether some of them might be removed, transformed, combined (ie, sum or difference of variables, or other ways of extracting new variables from the existing ones) etc.
- Here, there was no direct treatment of outliers. Some outlier treatment techniques might be useful during the feature engineering phase to make for more robust results.
- All regression models were ran over exactly the same, transformed dataset. It might be possible that applying different methods of feature engineering for different models could be better and lead to improved results.

- Other kinds of regression models might be tried, for example, Support Vector Machines or Neural Networks.
- Other techniques might be tried for combining models. For example, I can think of a combined grid-search and averaging technique to both set the individual models' parameters and the weights of the models on the averaging phase, so that each model's parameters will not necessarily be the same as if the individual model is found and fit “alone” on the data.

Personally, since I have professionally worked on some regression models for oil industry several, this Capstone Project was a nice opportunity to bring me back to some old days. Having worked on the past with very different tools, being now a first-time user of Python (and the scikit-learn library) and being far from a full-time programmer, this was quite a task for me. Dealing at the same time with a new language and the intricacies of scikit-learn and the Pandas framework were my main difficulties during this project.

In the end, I want to add that this model averaging method was something I thought about by myself, without looking at scientific literature or blog posts. I haven't read the Kaggle forums either or elsewhere to look for other ideas on how to solve this regression problem.