Predicting Rental Listing Interest

Udacity's Machine Learning Nanodegree

Capstone Project - Murray Lynch

Kaggle Competition: Two Sigma Connect: Rental Listing Inquiries

July 5, 2017

Note to Udacity reviewer

I have organized this report with the report template and rubric in mind. Instead of breaking up the iterative process into a separate parts, related work is grouped together. For example, discussions of "algorithms and techniques" are done across sections as contextually appropriate.

I went through the recommended structure so I didn't forget anything, and so it was easy to list items from the template under each combined section's heading. Perhaps all that wasn't worth the trouble, but oh well, happy reviewing!

Project Definition

Rubric sections included:

- **Definition**
- **Analysis: Algorithms & Techniques (partial)**

Overview

RentalHop is an online apartment rental listing for the New York City area. One of its differentiating features is its relevancy score, a “HopScore”, by which it sorts listings by default. They would like to harness data on their listings to improve their product in other ways, including fraud detection and quality control.

For this, Two Sigma, their data-focused management group, have partnered with Kaggle to hold a machine learning competition: Two Sigma Connect: Rental Listing Inquiries.

The dataset is available on the Kaggle competition page. A login is required to accept their terms & conditions.

Problem Statement

RentalHop has back-end functions that could benefit from reliable predictions of how much interest individual listings will generate. These functions are:

- Fraud identification
- Quality control
- Guiding owners and agents toward better listings

By applying a variety of machine learning techniques on rental listing data (price, location, etc.), an algorithm can “learn” complex patterns that correspond to levels of interest users will have in different listings. This model can then provide estimates of interest levels for new listings.

Specifically, the model will be developed to predict a categorical variable: ‘low’, ‘medium’ and ‘high’ interest levels. These are aggregates of the number of inquiries a listing receives while it is live on the site. An inquiry occurs when a user submits a form to contact the listing’s renter.

The model's development will utilize a broad set of techniques, their successful applications being the substance of this report. These are the families of techniques:

- Exploratory Data Analysis
    - Univariate and multivariate statistical analyses, data visualization and dimensionality reduction.
- Feature Engineering
    - Outlier detection and imputation, feature extraction (aggregation, dimensionality reduction, vectorized representations), feature selection, and feature scaling.
- Model development, selection and tuning
    - Model selection, hyperparameter tuning, model evaluation, validation and calibration (for probabilistic models).
- Presenting Results
    - Interpreting and assessing results with visualizations and other forms of communication.

The Evaluation Metric and the Probabilistic Approach

This project requires a probabilistic model allows for models that are responsive to changing decision-making environments. Levels of certainty are ranked and used for prioritizing one action over another.

In the case of flagging posts for potential fraud or quality issues, more certain predictions of a low interest listing can be prioritized over less certain ones. Different thresholds of certainty could be used to group listings together for different responses. In this case, those listings that are very likely to garner 'low interest' could have their accounts suspended until the matter is resolved. Listing that are likely, but less so, may allow the poster to maintain posting privileges.

These methods allow decision-makers to balance the quality of listings with poster relationships in an dynamic business environment.

The competition is evaluated with *multi-class logarithmic loss* (MLL), an error metric to be *minimized*. This is evaluation specification provided by kaggle:

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log(p_{ij}),$$

Where:

- N is the number of listings in the test set,
- M is the number of class labels (3 classes),
- log is the natural logarithm,
- y_{ij} is 1 if observation i belongs to class j and 0 otherwise, and
- p_{ij} is the predicted probability that observation i belongs to class j.

Essentially, it is the average log-likelihood of true labels across listings.

The logistic transformation in a logistic regression for binary classification can be generalized to multi-class classification problems using the softmax function. MLL is the corresponding generalization of logarithmic loss, a.k.a. logistic loss. In-practice, this loss is negated so that it fits into existing frameworks where metrics like accuracy are *maximized*.

MLL, in this particular case, is not just a measure of how well a model solves the problem like accuracy and precision, to name a few. It is also integral to the problem-solving technique. MLL is the basis of the objective function, whose optimization by a given model yields the solution, i.e. reliable predictions. Algorithms like gradient descent and gradient boosting work to minimize the loss function, in doing so, training a model toward its most probable parameters given the process by which the data is generated. The other part of the objective function is a regularization term that controls the model's complexity in an effort to learn more signal and less noise. In minimizing the objective function, MLL is itself improved while balancing the model's ability to generalize to unseen data, i.e. the goal of machine learning for classification.

MLL is also called *cross entropy loss*, introducing an information theoretic perspective to this optimization problem. Entropy in this context can be thought of as a measure of uncertainty of a probability distribution, that is how much confidence to have in it. Cross entropy is an entropy measure with respect to the true distribution of the target variable. With a perfect generalization score, the model would have perfectly captured this true underlying distribution. By optimizing the objective function, the model captures a best-approximation of the true distribution.

Feature Analysis and Engineering

Template sections included:

- **Analysis: Data Exploration**
- **Analysis: Data Visualization**
- **Analysis: Algorithms & Techniques (partial)**
- **Methodology: Data Preprocessing (partial)**

Dataset Overview

There are 49,352 listings in the training data, each of which is described by the following variables:

- `interest_level [str]`: The target variable, 'low', 'medium' and 'high' categoricals
- `listing_id [str]`: unique identifier for the listing
- `building_id [str]`
- `manager_id [str]`
- `latitude [float]`
- `longitude [float]`
- `price [int]`: in USD
- `created [datetime]`: date the listing was posted
- `bathrooms [int]`: number of bathrooms
- `bedrooms [int]`: number of bathrooms
- `street_address [str]`: does not include city or zip
- `display_address [str]`: the street address less street and apartment numbers
- `description [str]`: provided by poster
- `features [list of str]`: a list of features about this apartment
- `photos [list of str]`: a list of photo links. Image data itself will not used in this project.

Training data vs Test dataset:

To understand the provided datasets, it is important to understand how Kaggle competitions work. Two public datasets are provided:

- Training data, with target variable (49352 rows, 14 columns)
- Test data, no target variable (74659 rows, 13 columns)

Before the competition ends, submissions to the “private leaderboard” are based on predictions for the (public) test dataset, calculated by Kaggle’s servers. After the competition closes, the final score on the “public leaderboard” is calculated from an (private) unseen dataset 1.5 times the size of the provided test set. This presents a challenge of potential overfitting evaluations done before the competition closes.

Since the competition has ended at the time of writing this report, the public leaderboard score will be considered the final test set (instead of the private leaderboard whose score are no longer available). The test data from the train-test split of the labeled data will be treated as the holdout dataset which would have been the private leaderboard dataset. Using the private leaderboard dataset only once of the end of the project ensures that the final model is evaluated accurately, having not been fit in anyway to that data.

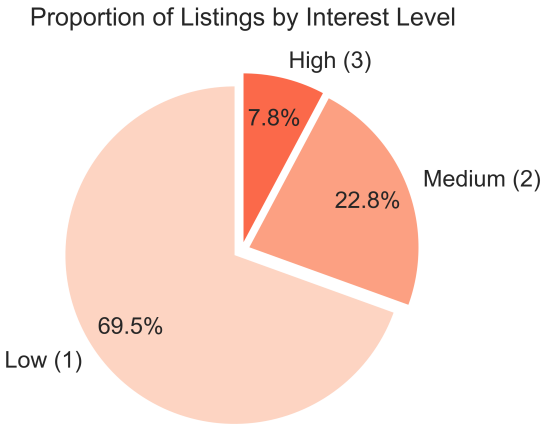
Both datasets run from April 1 to June 29, 2016, and based on Kolmogorov-Smirnov 2-sample tests on numeric variables, they were sampled from the same dataset. High p-values shown below indicate justify high confidence in this assumption.

	Price	Bathrooms	Bedrooms	Latitude	Longitude	Date by week
KS2 p-value	0.940	0.995	0.998	0.250	0.767	0.981

How interest-level data is generated and its implications for preprocessing

Missing values and outliers are explored in this section for both plotting and data quality purposes. However, changes to the dataset for training may be limited. Extreme outliers should be removed for generalized linear models (GLMs), but ultimately, for missing and incorrect data, these will impact the user’s engagement with a listing. Certain search queries will exclude these values altogether, and when they do show up, users are presented with these values.

Dataset Features

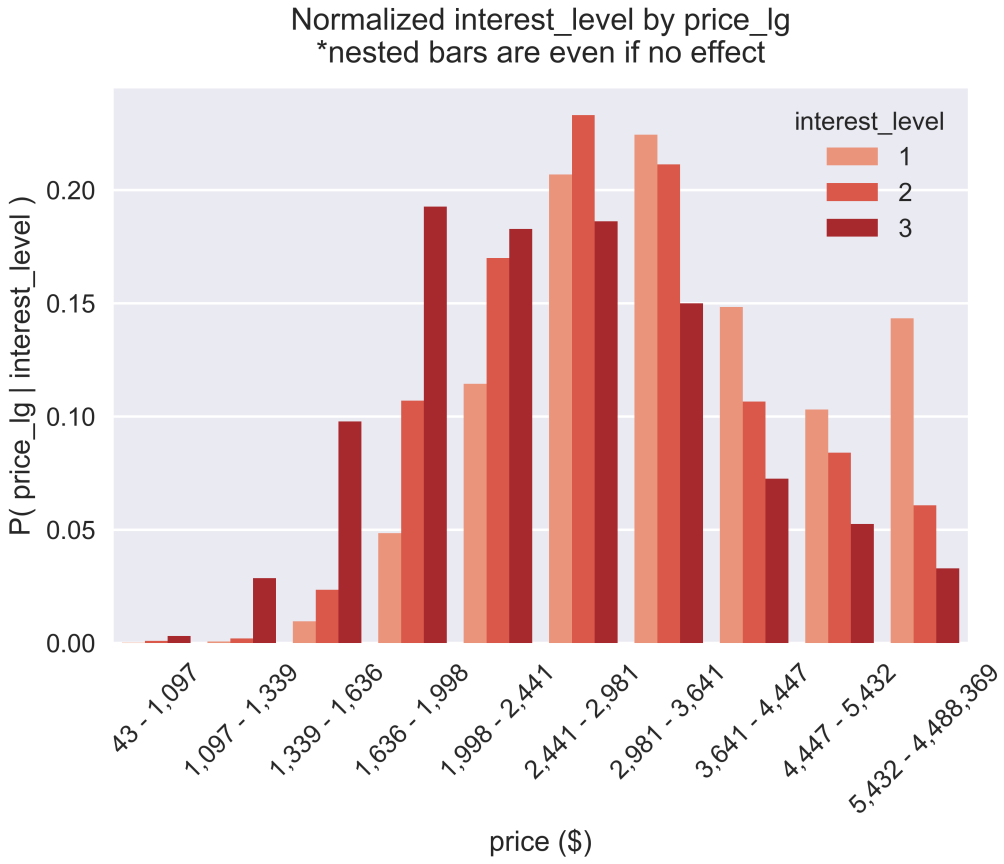


Interest Level

- 3 levels: low, medium, high
- Class imbalance not extreme enough to warrant down-sampling or other related techniques.
- It is sufficient to employ stratified sampling in conjunction with probabilistic models whose predictions are properly calibrated. that take into account imbalance is sufficient.
- *Transformation:* None

Price

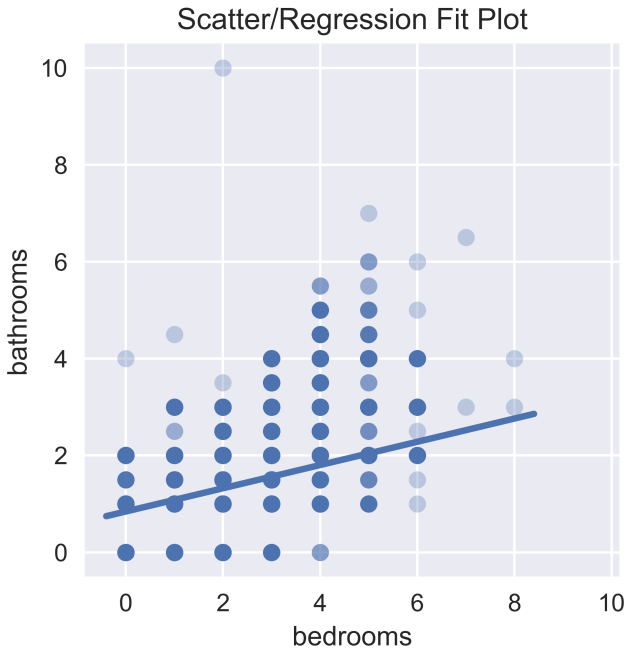
- Distribution: Log-normal
- Summary statistics, taken from logged distribution (exponentiated then rounded):
  - mean: $3,291; min: $43; 25%: $2,500; 50%: $3,150; 75%: $4,100; max: $4,500,000
- Large impact: as price increases, interest level decreases



- *Transformation:* Logged if data will be standardized as required for some non-tree-based algorithms.

Bedrooms and Bathrooms

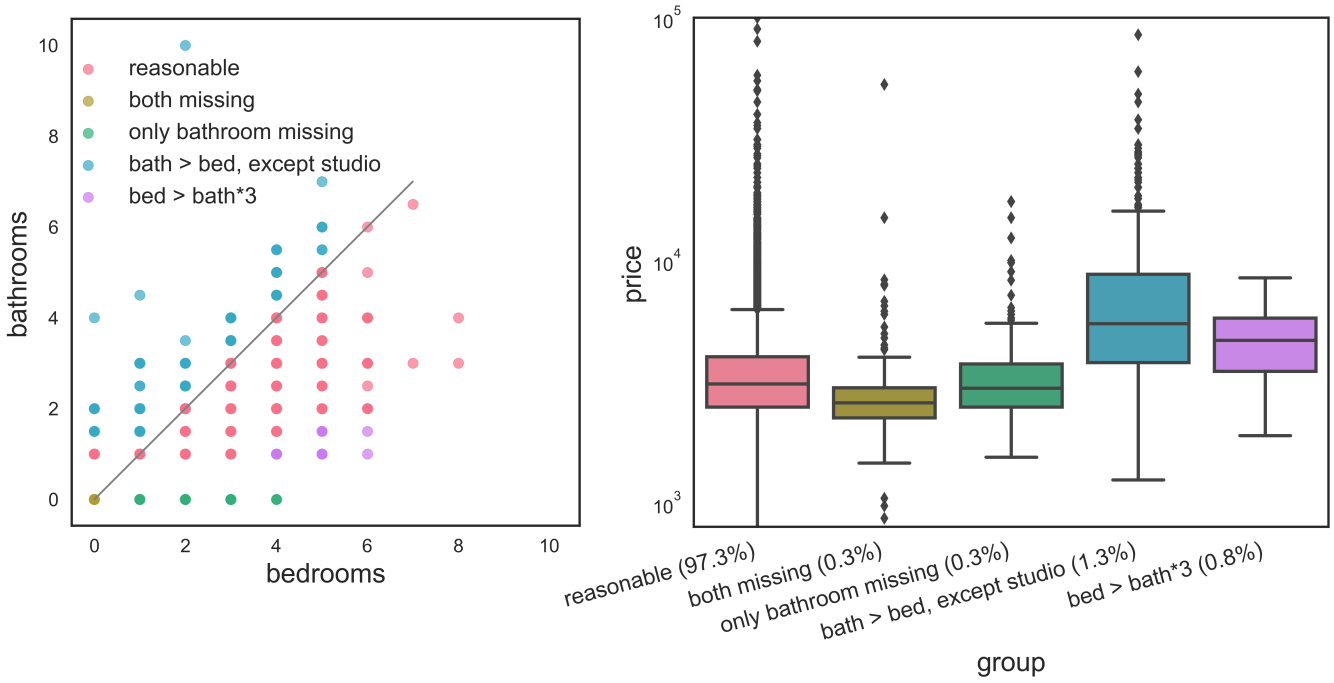
- Bedrooms and bathrooms, not unexpectedly, are fairly correlated, and as such their relationships can be used to assess outliers, missing or incorrectly-entered values.
- These are indications of potential input mistakes at listing creation time.
- Imputing suspicious values may improve model performance, but on the other hand, this data represents what the user sees it on the website, which has a direct effect on user interest level.
- An investigation into these two variables found 4 groups of strange bedroom/bathroom combinations.
- These are visualized and thoroughly discussed in the table and plots that follow.



Mutually Exclusive Groups	**Count**	**Median price***	***Possible Transformation***
Listings with no bedrooms nor bathrooms	156 (0.3%)	$2,595	Impute with corresponding median from whole dataset.
Listings with no bathroom, but one or more bedrooms	157 (0.3%)	$2,982	Impute bathroom value with median taken from listings with corresponding number of bedrooms (scatter plot above show a relationship between the two).
More bathrooms than bedrooms, except for studios	634 (1.3%)	$5,500	Do not impute. Given the higher prices of these listings, they are likely accurate.
Listings more than 3 times as many bedrooms as bathrooms	382 (0.8%)	$4,695	Do not impute. Given the higher prices of these listings, they are likely accurate.

*Compare to dataset median of $3,150, all means calculated from logged distribution

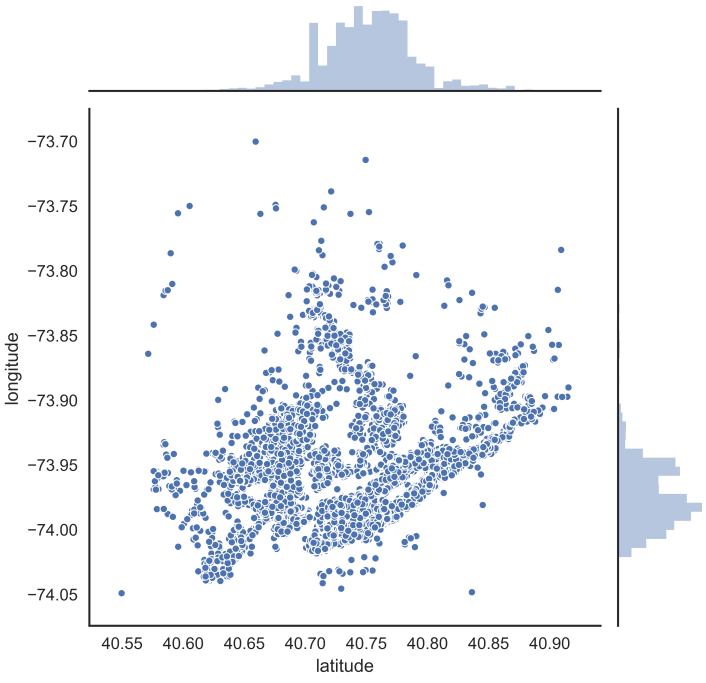
Strange bathroom/bathroom values



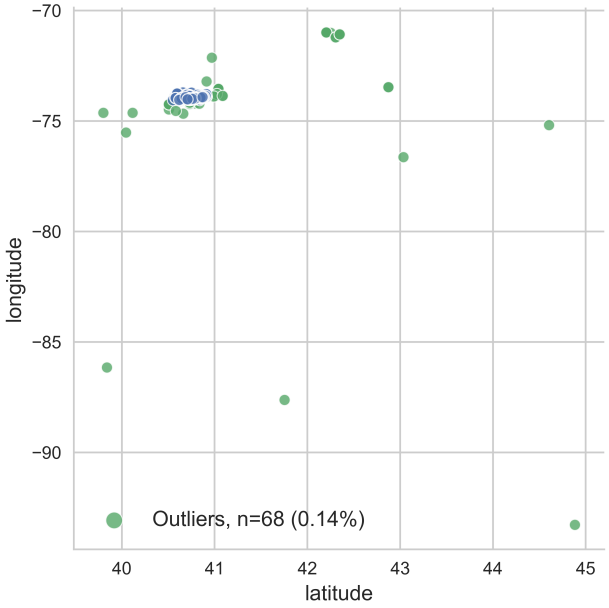
Latitude and Longitude

- There are 12 listings with zero geo-coordinate values, which can be imputed with mean values.
- Other outliers are removed for visualization purposes.
- *Transformation:* Impute handful of 0-value coordinates with mean.

Listing coordinates w/o outliers (NYC area)

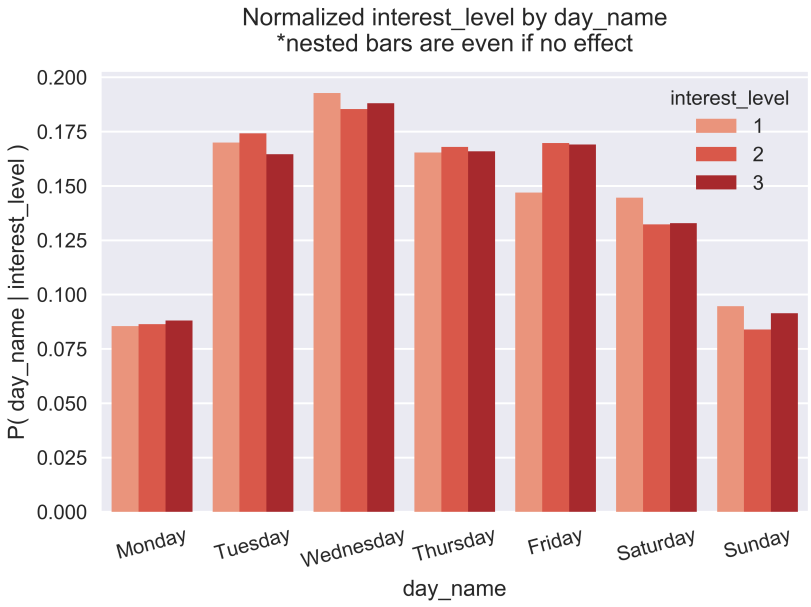


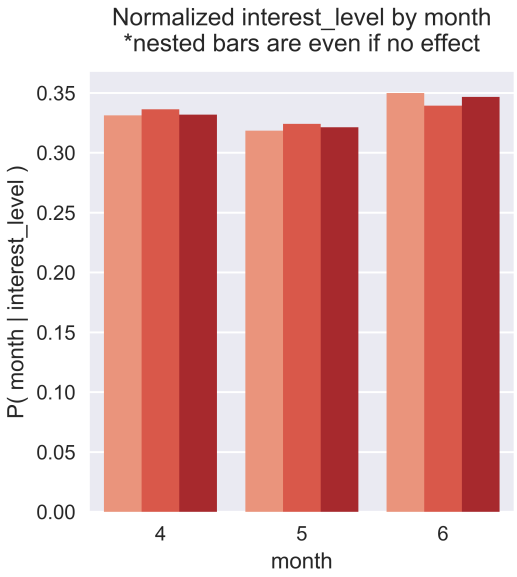
Listing coordinates w/o missing values (zeros)



Created (Dates)

- Date range: April 1 to June 29, 2016
- The bar plots below provide information on both the distribution of listings across day and month created, as well as the effect on interest level.
- The day has some effect and should be included in the training data, while the month is not relevant.
- *Transformation:* Make dummy variables for day names.





Count Features

Created with transformations:

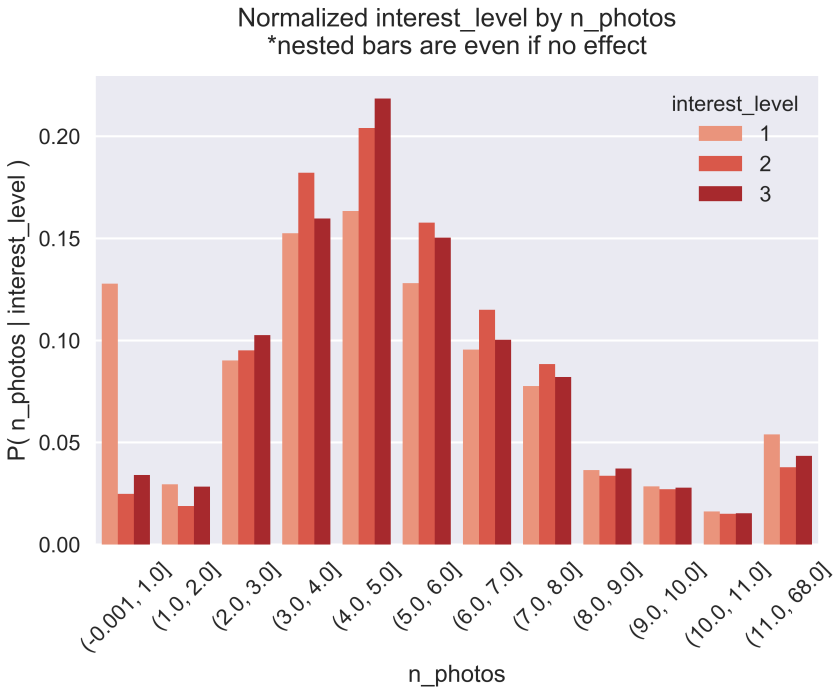
- Features of lists or paragraph text cannot easily be reduced to counts of their parts.
- Apply square-root if data will be standardized as required for non-tree-based algorithms.

Number of photos

The number of photos has some interesting properties as shown in the distribution plot below.

- The distribution is fairly normal with the exception of zero photos and a very long tail (compressed in the last bin).
- The most common number of photos is 5, which also has the most positive impact on interest level.
- As photos increases, returns diminish until at 9 photos, there is no longer an effect. One hypothesis is that managers with lots of experience tend to post a reasonable number of

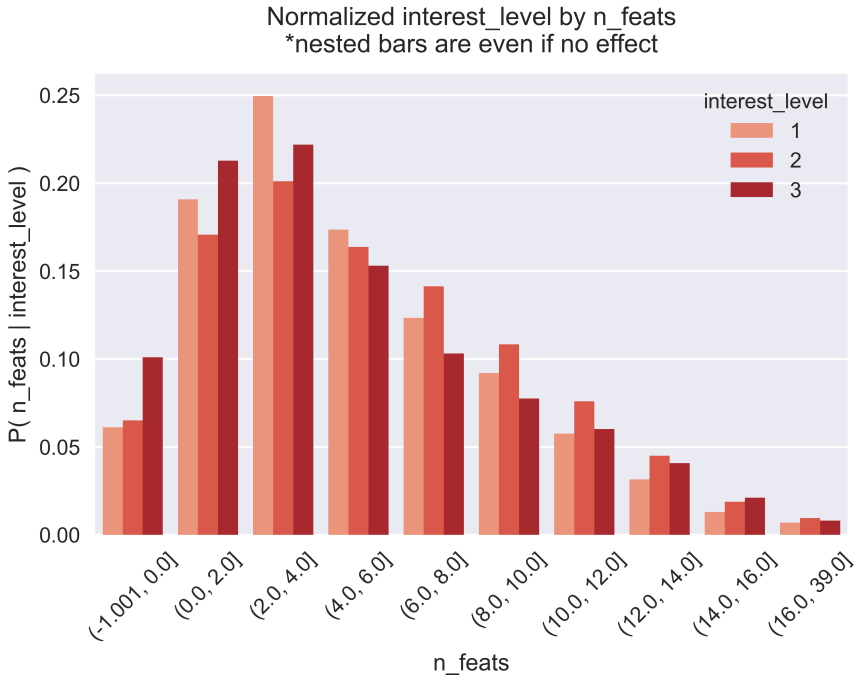
photos. Experienced managers will garner more interest as shown later in this report.



Number of features

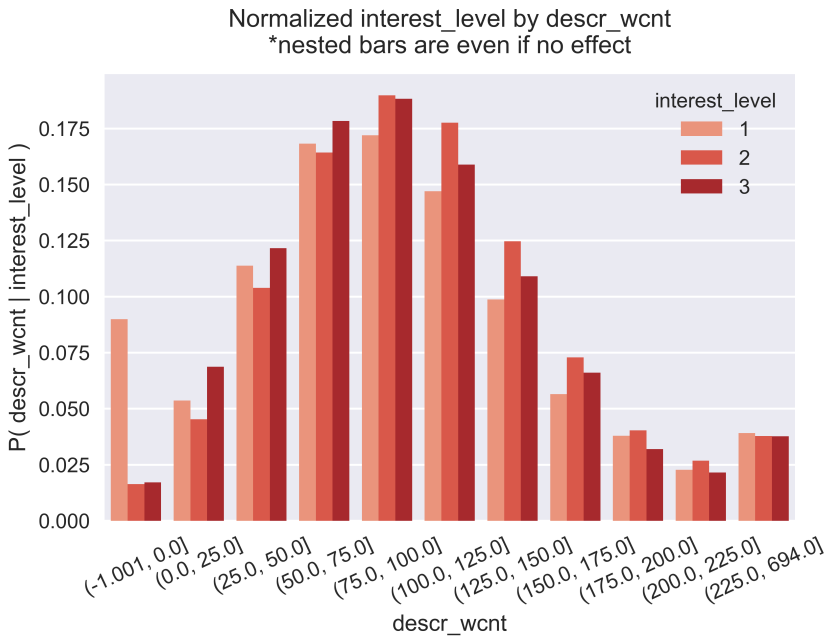
The number of features has some interesting properties as shown in the distribution plot below.

- The distribution is somewhat normal with the exception of zero photos and a very long tail (compressed in the last bin).
- The most common is the bin of 3 and 4 counts, which oddly has a higher proportion of high and low interest, taking from medium interest.
- The consequent medium count imbalance is distributed to counts of 5 to 12.
- Listings with no features have a significant positive impact on interest level.



Description word count

- The most common range of counts are between 50 and 125.
- While this variable has an impact on interest level, it is not clear what pattern it follows (with the exception of zero counts having significantly less interest).



Algorithm Selection

This section includes:

- **Analysis: Algorithms & Techniques, (partial)**

The specificity of the evaluation metric narrows the scope of algorithms to ones that can directly optimize multi-class logarithmic loss. Suitable algorithms are Gradient Boosting Trees and Logistic Regression. This will be spot-checked (quickly assessed before without much tuning). The algorithm that has the best metric will be further developed.

Gradient Boosting Trees (GBT)

Overview

- An ensemble learner that uses multiple instances of trained decision trees.
- CART (classification and regression trees) are trained sequentially, building an additive model in a forward stage-wise fashion, i.e. the final prediction is the sum of predictions .
- Like random forests, this is very effective at learning complex patterns through different subsets of the data.
- Unlike random forests, each subsequent tree fits to minimize a newly derived loss function based on the previous tree's residuals (i.e. the gradient).
- Instead of fitting each tree to predict the target variable, they are fit to predict its gradient (Source: Comment on StackExchange by Matthew Drury).
- The specific implementation of this algorithm used here is XGBoost. It differs from the traditional gradient boosting machine algorithm by incorporating regularization into each node splitting decision, making it very effective.

This algorithm is appropriate in this context because:

- It supports the multi-class log loss as part of its objective function.
- Hyperparameter tuning provides great flexibility in combating overfitting.
- It is computationally efficient allowing for easier tuning process.
- It allows for easy interpretation post-training with various measurements of feature importance.
- Using decision trees reduces preprocessing steps that are necessary with non-tree-based algorithms.

- It is both robust to outliers and does not need input to be standardized.
- This is because node splits use a line search algorithm that only considers the ordering of feature values and not distance between them and not the space between data points.

Input requirements:

- Covered in previous discussion on appropriateness for this problem.

Logistic Regression (multi-class)

Overview

- This is part of the family of generalized linear models that uses link functions to adapt to data that would have non-normal residuals with a least squares approach.
- To achieve prediction probabilities between 0 and 1, the logistic function is wrapped around the linear combinations of variables weighted by optimized parameters (or weights).
- With each step, weights are updated toward optimizing the objective function.

This algorithm is appropriate in this context because:

- Its hyperparameters tune easily.
- Its probabilistic outputs to do not need calibration (it is optimized for probabilistic output).

Input requirements:

- It is best practice is to standardize data to zero mean with unit variance. This can improve speed and likelihood of convergence by keeping step sizes for each weight relatively consistent.

Gradient Descent

This algorithm is used in many different algorithms, from neural networks, to linear and tree-based models. Logistic regression and BSTs have different implementations of the algorithm and should be discussed to better understand them.

Overview

- This is an approximation technique for convex optimization, minimizing convex objective functions (like regularized MLL).
- This is the only computationally feasible method of optimizing functions that cannot be reduced to a closed-form like square loss for a simple linear regression.
- The algorithm works by updating the model through its parameters, taking a step at each iteration down a convex curve toward its minimum, “descending” toward the trough.
- The direction of this update is determined by the negative gradient of each of the algorithm’s parameters, those being coefficients of linear regression and tree structures of GBT. The size of this update will be controlled by the “learning rate” hyperparameter.

GLMs vs boosting models

- GLMs apply this technique by updating the model’s parameters, in this case its weights or coefficients.
- Gradient boosting, on the other hand, treats tree structures as a parameters. Instead of updating all *weights* at each iteration as in GLMs, the additive model is updated by *adding a new tree* at each iteration.

Model Development

This section includes:

- **Analysis: Benchmark**
- **Methodology: Implementation**
- **Methodology: Refinement**
- **Results: Model Evaluation and Validation (partial)**
- **Results: Justification (partial)**

All learning and transforming algorithms use scikit-learn's implementation, exceptions being GBT which uses XGBoost's and custom transformers which are specified in this report and included in the accompanying code.

References:

Scikit-learn: Machine Learning in Python, Pedregosa *et al.*, JMLR 12, pp. 2825-2830, 2011.

Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In 22nd SIGKDD Conference on Knowledge Discovery and Data Mining, 2016

XGBoost originates from research project at University of Washington, see also the Project Page at UW.

Preprocessing Implementation

Input features:

- Basic features: price, bedrooms, bathrooms, latitude and longitude.
- Count features: number of photos, number of features, and description word count.
- For LogisticRegression, all features are standardized after any necessary power-transformations (discussed in the earlier section dealing with features).
- Training set consists of 75% of the data.

Custom transformation were necessary for techniques previously discussed feature-by-feature. These are provided in the preprocessing.py module that provides more detailed documentation with docstrings and comments where appropriate.

Stateless custom transformers:

- `LogTransformer`
- `SqrtTransformer`
- `LenExtractor`
- `WordCntExtractor`
- `DayBinarizer`
- `ItemSelector`
    - Adapted from: http://scikit-learn.org/stable/auto_examples/hetero_feature_union.html

Stateful custom transformers:

- `BedBathImputer`
- `LatLongImputer`

Stateless vs stateful transformers

Some transformers store data taken from the training data by the fitting algorithm. That data is used when applying the transformation to the training *and* testing data. A classic example is scikit-learn's StandardScaler. When fit, it computes and stores mean and standard deviation used when transforming the data.

If the StandardScaler instance used data taken from the test data when transforming the test data, this would cause 'leakage' of test data into training process. Subtle patterns in the test set would seep into the algorithm's learning process, causing overfitting. Leakage can also occur if results from the test data are paid too much consideration during model development. Transformers whose transform call uses results of its fit call are called *stateful*.

An example of a *stateless* transformer is one that applies a logarithmic function to each value of the passed dataset. Nothing is stored in the instance, and thus there is no conduit for any leakage from the test data.

Performance optimizations

After setting up and running a meta-transformer with FeatureUnion's and Pipeline's with all necessary transformer, performance analysis revealed very costly regular expression

evaluations. These calls are made during the `WordCntExtractor` transformation on the description feature.

Because this transformer is stateless, it is not necessary to include it as part of the meta-transformer used for cross-validation (GridSearchCV), where it is called on each fold for each hyperparameter combination. Instead, it can transform the description feature on the entire dataset before performing the train-test split.

This, and other stateless transformers, were transferred from the meta-transformer to a function called `feature_prep` (also in preprocessing.py). Another benefit of this change is a greatly simplified meta-transformer. In fact, the XGBoostClassifier does not need one and can be passed directly to GridSearchCV, while Logistic Regression is put into a Pipeline with only the stateful StandardScaler preceding it.

Benchmark

The benchmark is done without a learning algorithm. This entails calculating the negative log loss between the true classes and uninformed priors based on class probabilities across the entire dataset. Predicted probabilities to be scored are simply the distribution of labels, i.e. low=0.69, medium=0.23, high=0.80.

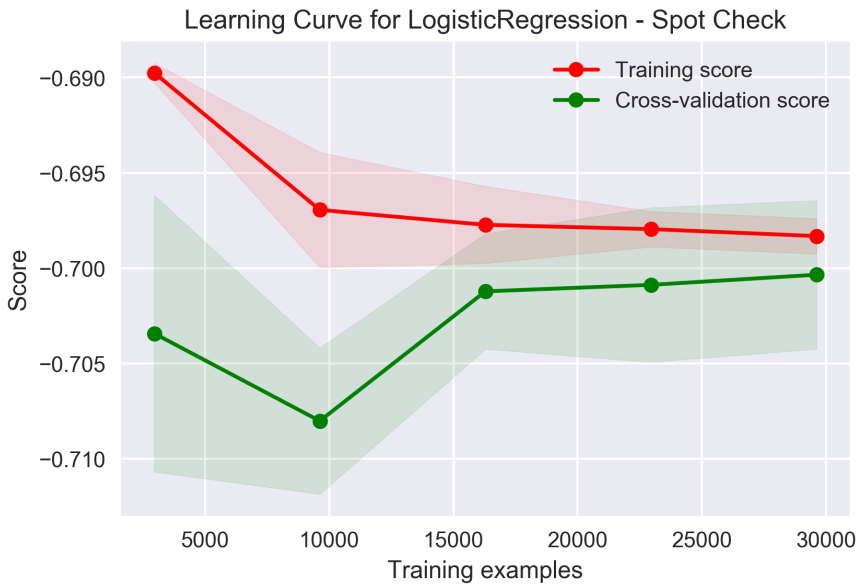
➡ Basic label distribution benchmark: 0.7886 (MLL)

Spot-checks

XGBoostClassifier and scikit-learn's LogisticRegression will be spot-checked, i.e. assessed on initial cross-validation scores from a set of reasonable parameters. These results will determine which algorithm will be further analyzed and developed.

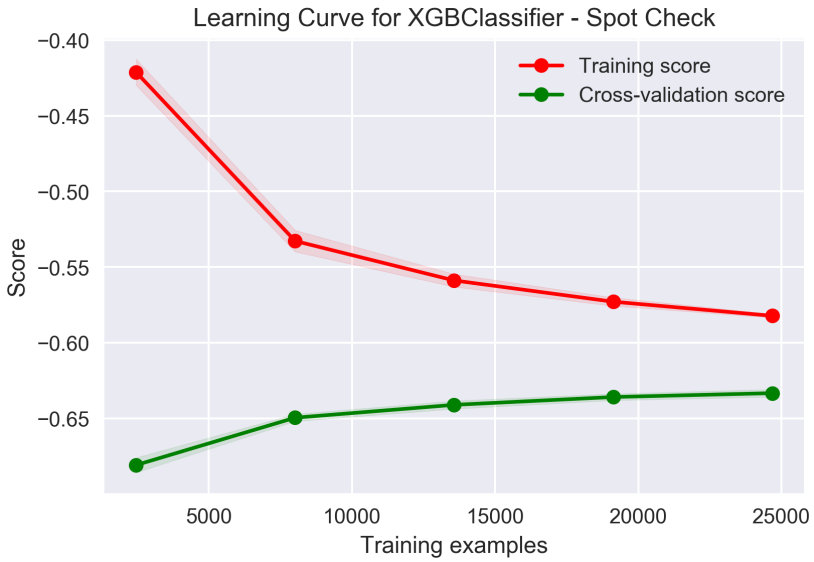
Algorithm setup and results:

LogisticRegression **- CV'd with GridSearchCV (incl. a bit of tuning)**		**XGBClassifier** **- CV'd with native xgb.cv**	
multi_class= **'multinomial'**	As opposed to one-vs-rest, which does not directly optimize multi-class negative log loss.	**objective=** **'multi:softprob'**	This uses the softmax function for the GLM transformation that provides probabilistic output.
solver= **'lbfgs'**	Compatible with 'multinomial', appropriate for large (but not very large) datasets.	**n_estimators=400** **i.e. the number of trees/rounds of boosting**	Guessed 200 based on the size of the data set (35k in the training data). The 'boosting' curve below justifies this choice.
warm_start= **False**	This uses data from previous fit for new fit's initialization, making it run faster. Set to False because it is ignored with GridSearchCV	**learning_rate=0.015** **Rule of thumb is (2 to 10) / num trees. Source:** **Presentation by Owen Zhang**	With each boosting round, or "step", the contribution of the next tree is decayed. Similarly to gradient descent in logistic regression, these steps must shrink to allow the algorithm to converge.
random_state= **42**	So that results are exactly reproducible.	**seed=** **42**	So that results are exactly reproducible.
C=9 evenly spaced values between -4e10 and 4e10	Hyperparameter range to search. Regularization strength (lower increases impact of regularization term)	**max_depth=6 (default)**	Once the tree-growing algorithm is 5 nodes deep, it cannot grow any more. A reasonable default.
Spot-check results *Pipeline used to include StandardScaler for LR only	CV score: **0.7003** Train score: **0.6984** *** For parameters: *** lr_clf__C=0.1	**Spot-check results**	CV score: **0.6346** Train score: **0.5816**



This learning curve indicates an algorithm that will generalize well to unseen data. This is impressive given it is barely tuned. However, this indicates less opportunity for performance gains through the model itself.

On the other hand, GBT present lots of opportunity to improve its already-superior evaluation metric performance. The divergent CV and training scores as boosting rounds approach 200 in the boosting curve indicate overfitting. And with a plethora of knobs to play with, this algorithm can be refined.



Model Refinement, Validation and Evaluation

Stage	Details (hyperparameter, etc.)	XGBoost (rounded)	Change
Benchmark	See Benchmark subsection above	MLL: **-0.7886**	**0.7886**
Spot-check	Simple grid search without any imputations. See spot-check section for more details,	CV score: **-0.6346** Train score: **-0.5816**	CV score: **0.1540** Train score: **-**

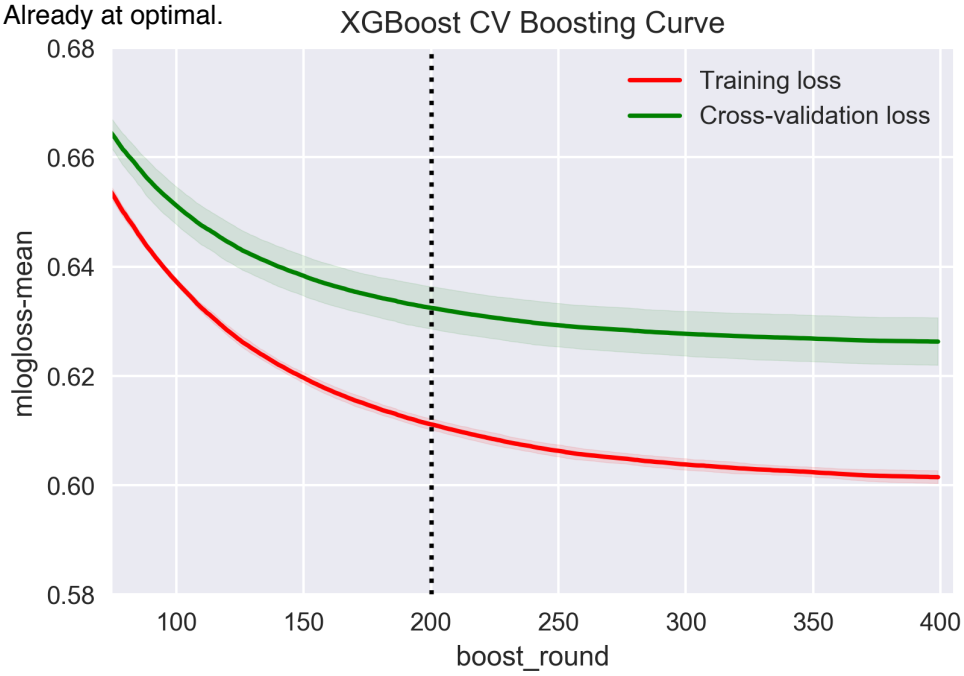
Stage	Details (hyperparameter, etc.)	XGBoost (rounded)	Change
1: XGBoost's missing value handling	Imputation strategy: - Impute missing values for features: bedrooms, bathrooms, latitude, longitude. - LatLongImputer(how=-999, broad=False) - BedBathImputer(how=-999) - See code for implementation details derived from earlier analysis. - Imputation value of -999 that is passed as an argument to the classifier.		

	Using hyperparameters from spot-check (except missing):
	
	`n_estimators=400`
	`learning_rate=0.015`
	`max_depth=6`
	`missing=-999`
		CV score: **-0.6327**
		Train score: **-0.5916**
			CV score: **0.0019**
			Train score: **-0.0100**

	Conclusion: While the CV score barely changed, the training score decreased, implying the model suffers from the same bias but with less variance. After tuning stage, this will be compared with other imputation strategies.		
2: Tune max_depth, gamma and colsample_bytree	Since the model suffers from overfitting, these are the parameters to focus tuning on. Source: XGBoost docs, Notes on Parameter Tuning		

	`objective='multi:softprob',`
	`n_estimators = 200,`
	`learning_rate=0.05,`
	
	`max_depth=5,`
	`gamma=5,`
	`colsample_bytree=.8,`
	`missing=-999,`
		CV score: **-0.6346**
		Train score: **-0.6120**
			CV score: **-0.0019**
			Train score: **-0.0204**

	Conclusion: With the parameters which provide the best bias-variance tradeoff, the CV score remains essentially constant (the small decrease is insignificant), while the gap between CV and train score decreases again.		
3: Tune n_estimators	Tuning this parameter also provides an opportunity to lower overfitting, but also underfitting.		
	No change	No meaningful improvement.	No change

Stage	Details (hyperparameter, etc.)	XGBoost (rounded)	Change
	Already at optimal. 		
4: Tune max_depth and min_child_weight	No change	No meaningful improvement.	No change

5: Reassess XGBoost's missing value handling	The following imputation strategies are assessed in combination:
	- Bedrooms and bathrooms:
	  - Use various medians depending on the missing value's criteria
	  - -999 impute value for all missing value definitions
	- Latitude and longitude:
	  - Simple mean imputation
	  - -999 impute value for all missing value definitions
	  - If broad parameter is true, missing values are defined as values outside of range used for scatter plot. Otherwise, 0 values and a handful of very extreme outliers will be considered missing.
	
	See code for implementation details derived from earlier analysis.		

	No change	No meaningful improvement.	No change

Results

Final model:

```
objective='multi:softprob',
n_estimators = 200,
learning_rate=0.05,

max_depth=5,
gamma=5,
colsample_bytree=.8,
missing=-999,
```

Sensitivity Analysis

To assess the robustness of the final model, the logged price feature was distorted with random noise.

Noise range: 0.0052 to 0.0079

Hold-out w/o noise: **-0.6290**

Hold-out with noise: **-0.6318**

Difference: **0.0028**

This change in score is reasonable for a robust model.

Hold-out and Test Results

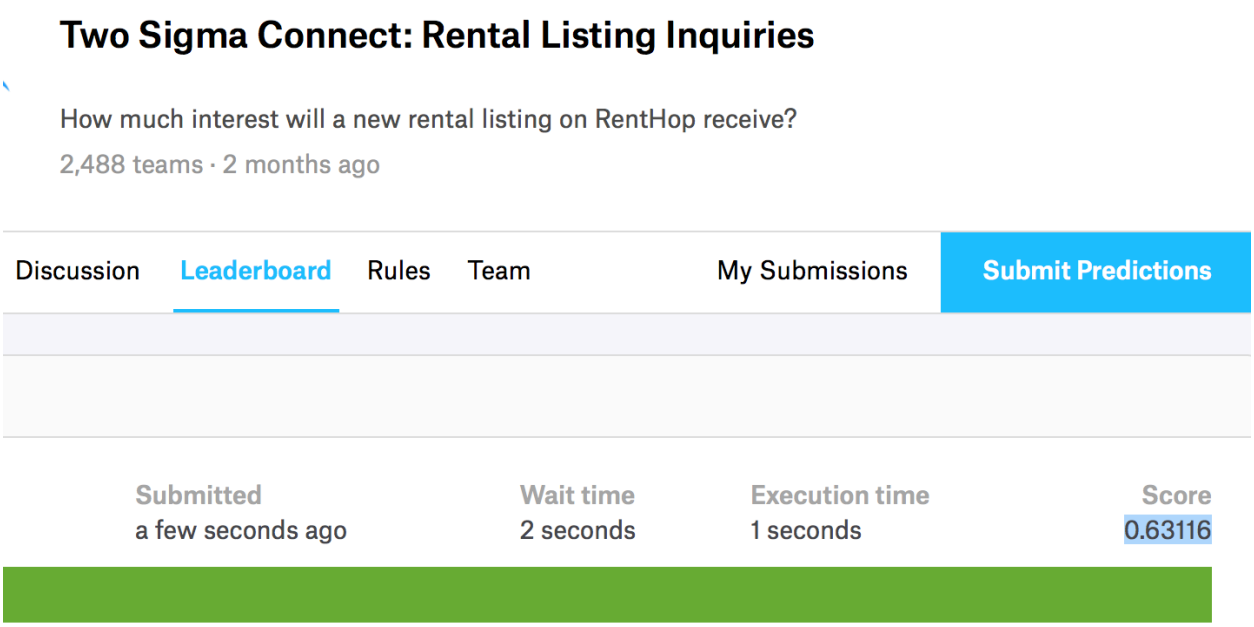
CV score: **-0.6347**

Holdout set score: **-0.6290**

Difference: **0.0057**

Test set score: **-0.6312**

Difference: **-0.0035**



Model Justification

All of the tuned parameters work against overfitting. They constrain the model complexity in light of the significant different between training and CV scores from the spot-check, an indicator of overfitting. Details of the model and its hyperparameters are provided in the refinement section.

Compared to the benchmark and spot-checks scores, this is an effective model. The learning curve comparison plots drive this point home.

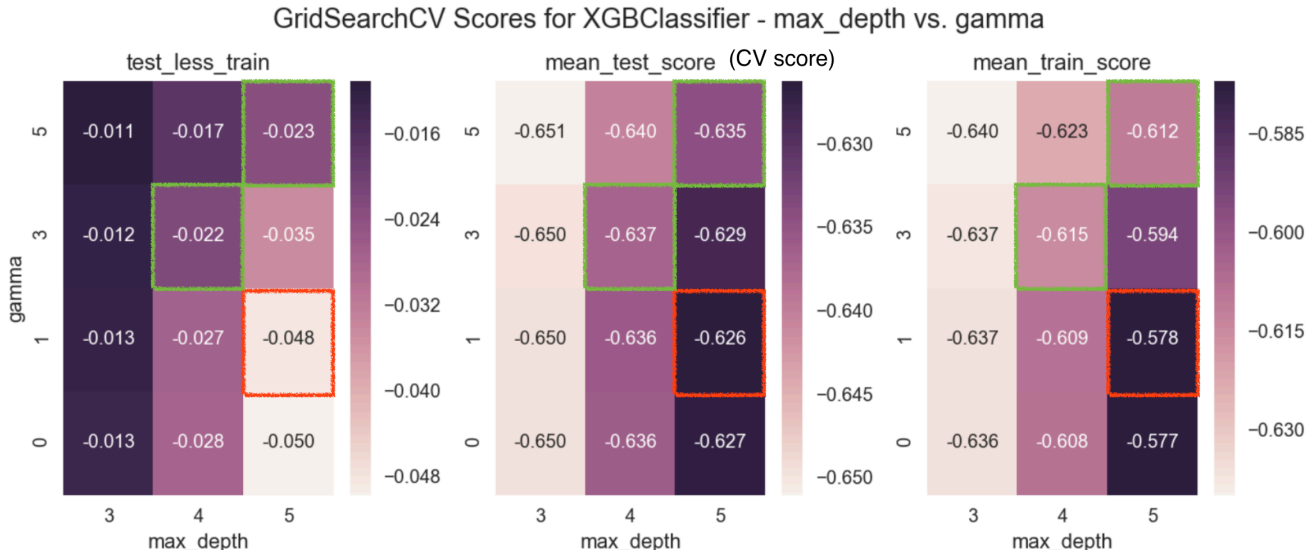
Conclusion

Free-Form Visualization

The results of a hyperparameter grid search can be misleading. The built-in assessment of the best estimator only considers the CV score, and which estimator has the highest one. However, the best estimator is one that best balances bias and variance. Since the XGBClassifier had significant issues with variance, the estimator with the best CV score oftentimes had the largest gap between it and the training score.

```
Best grid CV score: -0.6262 ± 0.0023 (mean ± std. dev.)
Train score:        -0.5777 ± 0.0022 (mean ± std. dev.)

*** For parameters: ***
gamma=1
max_depth=5
```



The above plot shows the effects of different hyperparameters combinations, and allows a determination of which offers the best bias-variance balance. The leftmost heatmap shows the differences between training and testing scores (darker indicating a small difference, i.e. less variance).

This graph shows the effects of gamma and max_depth:

- As gamma increases, penalizing complexity, the training score decreases and the CV score increases.

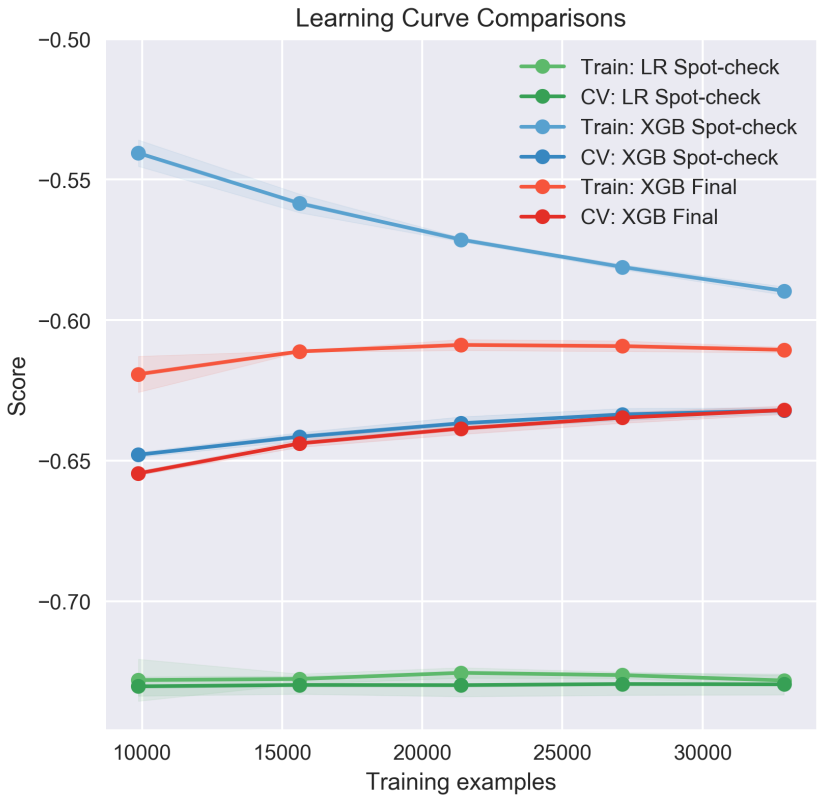
- As max_depth increases, adding complexity, the training score increases and the CV score decreases.

Highlighted in red is the estimator with the best CV score, and that returned by GridSearch's best_estimator attribute. This estimator has the 2nd highest gap to the training score, suffering from the second-most highest variance.

The scores highlighted in green show estimators with roughly the same best balance between bias and variance. Looking at the graph, these balance the darkness of the CV score blocks with the darkness of the score difference blocks, looking to maximize the two.

Reflection

The process detailed in this report moved from feature analysis and engineering through model development. Features were analyzed for relevance and engineering for quality. Model development consisted of spot-checking two suitable algorithms, logistic regression and gradient boosting trees. The GBT model performed significantly better than the LR one, and was selected for further development.

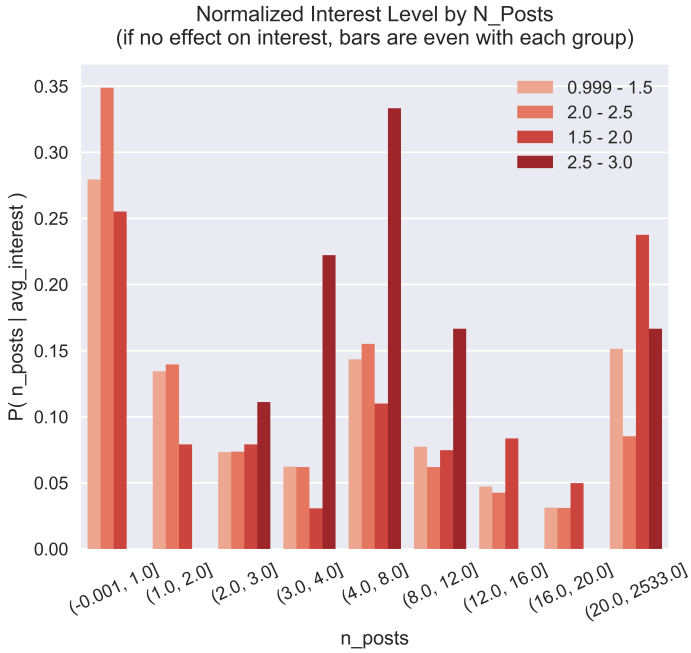


The refinement of this model focused on lower the variance of the model, since that was its glaring deficiency. Through different hyperparameter combinations in the tuning stage, it was possible to significantly decrease the variance of the model, while there was no improvement possible for its bias.

Improvement

There are multiple avenues where improvements to this project could be made, all involving feature engineering. Using descriptions and feature lists with natural language processing techniques could significantly improve the final score.

Another improvement just using techniques in this report would involve aggregating the number of listings per manager. The plot below show significant effects of manager activity and listing interest levels.



The stateful transformer algorithm that extracts this aggregate feature follows the steps:

1. Fit the training data
    - Store the count values of number of posts per manager_id as parameters.
2. Transform the training data
    - Normalize counts by number of listings in the training data.
    - Merge these values onto listings by their corresponding manager_id.
3. Transform the test data
    - Add stored count values from fit with counts of manager postings from test data.
    - Normalize by total number of listings in both datasets
    - Merge these values onto listings by their corresponding manager_id.

Normalization ensures transformations keep the same scale for counts after adding data stored during fit.

This transformer is analogous to an instance-based learning algorithm like kNN. While kNN stores data from training to make predictions, this type of transformer uses stored data for transformations.