Regression techniques comparison for house price prediction

Andrej Hafner
ah4910@student.uni-lj.si
Faculty of Computer and Information science
University of Ljubljana

Abstract—Estimating the price of a house can be a hard task. There are a lot of parameters to consider, which are connected in ways humans don't usually see. Regression models are used in order to predict the selling price from the 79 features that describe a house. Using engineered features we train regularized models, support vector machine, random forest and gradient boosting methods to find the best performing one. We show that gradient boosting methods are the best for the task, specifically model LightGBM, which achieves the lowest RMSE error.

Index Terms—housing market, regression techniques, price prediction, feature engineering, linear models, ensemble methods

1 Introduction

Estimating the value of a house we want to sell is a hard task. There are a lot of parameters, which impact the selling price. It usually requires a real estate agent that is knowledgeable in the house market of a certain area to give an estimation of the price. But this requires the person selling the house to pay an agency to give a price estimation, which not all people are ready to afford. Those are left with their own knowledge and can overestimate or underestimate the price, leaving them with an unsold house or with less money than they deserve. What if we could train a model to predict the price of the house? This would solve the problem for anyone selling the house, if the person had the data that the model required.
We can use regression machine learning models to try and predict the price. Models can be better than humans at estimating the price, since they can find hidden relations between the parameters, which humans can't see. For this task we'll use Ames housing dataset [1], on which we'll train different regression models. We'll start by analyzing our dataset in order to see what information it contains. Next we'll clean the data, since almost no dataset gathered in real life is perfect. Then we'll look at our features and put them in such form, that they'll be suitable for training. We continue with the description of regression models used. Selected models mostly use different regression techniques, thus we'll be able to see which are the most useful for the task. In the end we evaluate the models with two metrics using cross-validation, followed by the analysis of the results.

2 Related work

In this chapter we go through some of the related work that has been done in the field of house price prediction.
Authors of article [2] analyzed the impact of school characteristics on the house prices. Data used was from Chicago from the years between 1987 and 1991. They found that individuals are willing to pay more for a house that is in proximity to a school with higher scores on standardized tests. Different linear models were used for regression. Authors of article

[3] compared hedonic regression and artificial neural networks for predictions on house prices in Turkey. Their dataset contained 46 variables describing characteristics aspects of the house. Looking at the result of the hedonic model, they find that the pool, type of house, number of rooms and the house size have the biggest impact on the selling price. Artificial neural networks performed almost three times better than the hedonic model in terms of root mean squared error.
Article [4] describes using a combination of ridge regression with a genetic algorithm to predict house prices on the Korean real estate market. Genetic algorithm was used to find the optimal predictor variables to be included in ridge regression to minimize the root mean square error. In their evaluation the combination outperformed artificial neural networks and normal ridge regression.

3 Data preparation

In this chapter we first describe and analyze the Ames housing dataset [1]. Next we clean the data, handle missing values and select relevant features. Finally we engineer new features from existing ones in order to further improve the performance of our algorithms.

3.1 Dataset

Ames housing dataset [1] is a record of describing property sales that occurred in Ames, Iowa between 2006 and 2010. All of the sales are residential houses. If the property changed ownership multiple times during the data collection period, only the most recent sale was kept. It contains 2930 entries with 80 features. Most of the features are the type of information that a buyer would like to know when selecting a property (e.g. square footage of the house, square footage of the whole property, year that the house was built, number of bathrooms, size of the garage, etc.). There are 20 continuous features, mainly describing various area dimensions. Next we have 14 discrete features, which quantify occurrence of different items and parts of the house and. Lastly there are 23 nominal and 23 ordinal features, identifying different conditions or materials or describing overall quality. Our target variable is going to be *SalePrice*, which is the price of the house in US dollars.

3.2 Cleaning data

Like almost any real life dataset, ours is imperfect too. It contains missing values in 19 features. For categorical features we solve this by filling the missing values with their default values as described in feature description (e.g. we have a feature named *KitchenQual* which describes the overall kitchen quality. We fill missing values with value *TA*, meaning that the kitchen has average quality). For numerical values we will the missing values with the mode or median of the non missing values from that feature. We could remove the entries when the missing percent is low, but since the dataset is not very big, we prefer to keep entries and fill in the missing values. For some features values are missing because that element of the house is missing. For example, *GarageCars* and *GarageArea* have missing values when there is no garage on the property. We fill missing values like this with zeros, which marks that the property has no garage. For categorical variables that we don't know how to reasonably fill, we use Python programming language value *None* and for numerical we use zeros.
We handle outliers in the data later, before inputting the data into the model for training. For each feature independently we remove the median and scale the data according to the interquartile range (between the 1st and 3rd quartile). This is better than commonly used subtraction of mean and scaling to unit variance, since it's more robust to data with outliers.

3.3 Feature engineering

We start by calculating the correlation between all of the features. Correlation is a statistical measure that expresses how much two variables are linearly related, meaning that they change together at a constant rate. Pearson

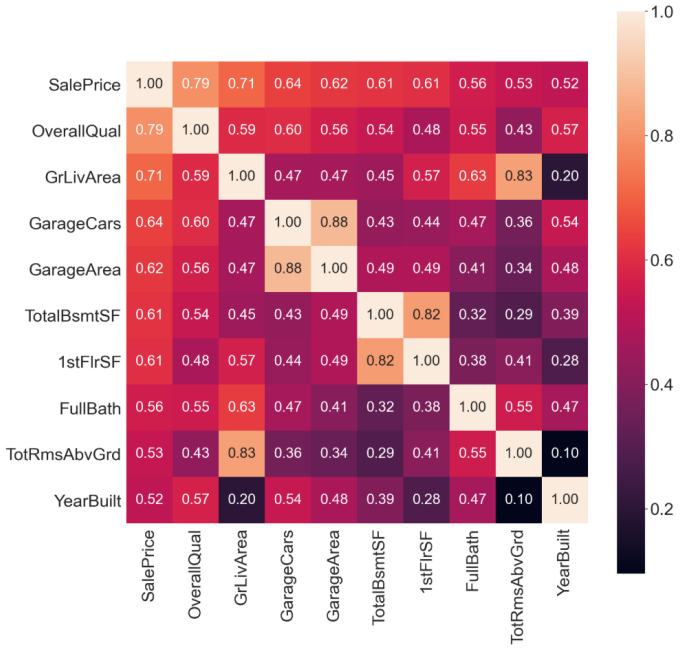

Fig. 1. Correlation matrix between the top 10 features that have the highest correlation to target variable *SalePrice*. Feature *GrLivArea* stands for above ground living area square feet, *TotalBsmtSF* for total square footage of the basement, *1stFlrSF* for 1st floor square feet and *TotRmsAbvGrd* for number of rooms above ground, excluding bathrooms. Rest of the feature names are self-explanatory. We can see that some of the indepentent feature have very high correlation, which can cause multicollinearity. This is bad because independent variables are correlated, thus no longer "independent". Consequence of this can be that the model can no longer evaluate dependence of a single independent variable to the the dependent variable, independently of others.

correlation coefficient is defined in equation (1), where X and Y stand for two of the features.

$$\rho = \frac{cov(X,Y)}{\sigma_x \sigma_y} \tag{1}$$

Next, we select the top 10 features that have the highest correlation to the target variable *SalePrice* and present their pairwise correlation in the correlation matrix (figure (1)).

In the first row (or column) we can see the features that have the highest correlation to target variable. The top two are *OverallQual*, which tells us the overall quality of the house and *GrLivArea*, which tells us the total square feet of the house above the ground. These results could be expected, since they are some of the main factors considered when purchasing a house. Another important thing to note is that some of the independent features are also highly correlated. *GarageCars* and *GarageArea* have very high correlation, which is to be expected, since more cars require more space. Another pair is *TotalBsmtSF* (basement square footage) and *1stFlrSF* (first floor square footage). Again we can explain this by assuming that an average house has a basement of the same size as the 1st floor of the house. Correlations like these can cause multicollinearity in the data, which can badly effect models, because they can no longer independently evaluate each independent variable and its relation to dependent variable. As we will later see, some of the models can avoid this, while others can't.

Another thing we have to consider is normality in the data. This tells us how similar is the data distribution to normal distribution. Most machine learning models don't perform well with data that is not normally distributed, hence we have to handle this as well. We can check if a feature is not normally distributed by calculating it's skewness. Skewness is the degree of distortion from the symmetrical bell curve or the normal distribution. A symmetrical curve has zero skewness. Positive skew is when the curve leans to the right (top plot in figure (2)) and negative when it leans to the left. We define moderate skewness when then absolute value of skewness is above 0.5. For independent features that have absolute skewness above 0.5 we apply power transform, specifically Box-Cox transformation [5]. This is a data transformation, which stabilizes variance and makes the distribution of the data more similar to normal distribution. We use natural logarithm transformation for our target variable, to fix it's skewness. It's similar to Box-Cox transform, but we don't use it because of reverse transformation after prediction. Because of this we train our models on the natural logarithm of *SalePrice*. After the prediction we can transform the value back to original *SalePrice* by applying a natural exponent function to it. In top graph of figure (2) we can see the distribution of *SalePrice* before applying the transformation. It's positively skewed and deviates from normal distribution. In the bottom image of figure (2) we can see the *SalePrice* after natural logarithm transformation. It's closer to a normal distribution (shown

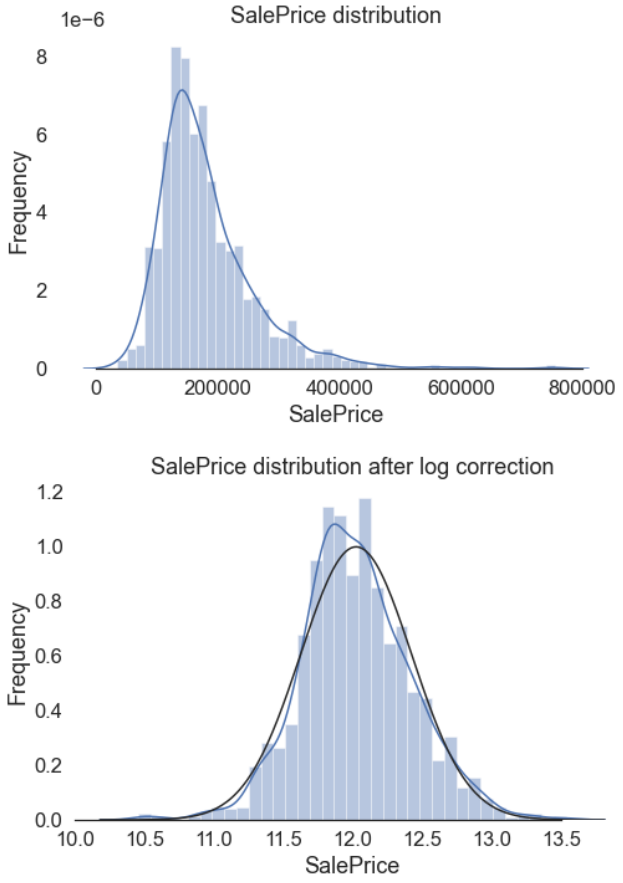

Fig. 2. The top plot contains the distribution of *SalePrice* target variable. It is positively skewed and not normally distributed. In the bottom plot we can see the target variable after applying the natural logarithm transformation, which makes it much more similar to normal distribution, shown by the black line. Normality of features is important for improving the performance of machine learning models.

by the black line), thus better for our models.

We continue with engineering new features from existing ones. This is done in order to extract new domain information, with which machine learning models can achieve better performance. Multiple new features are created, first one being *TotalSF*. It's a sum of *TotalBsmtSF*, *1stFlrSF* and *2ndFlrSF*, giving the total indoor square footage of the house. Next feature is *Remodeled*, which tells us whether the house was remodeled or not (when *YearRemodAdd* is not equal to *YearBuilt*). The fact that a house was remodeled can have a substantial impact on the price as the machine learning models wouldn't get this information from the two numerical features. Feature *TotalBathrooms* combines 4 features that give us information about the bathrooms inside the house. Finally, we have 5 new features, based of numerical features: *HasPool*, *Has2ndFloor*, *HasGarage*, *HasBasement*, *HasFireplace*. All of them are binary features, which are very important when considering buying a house. The information is already hidden in the existing numerical features, but giving it explicitly to the model can improve performance.

In the end we have to convert all the categorical variables into dummy variables. We do this using one hot encoding. For a each categorical variable we create a new variable for each of the possible categories (e.g. if a feature has 3 possible categories, we create 3 new features and remove the original one). The value of new features is then either 0 or 1, denoting the presence or absence of that category. This is done because some machine learning models can't work with non-numerical data. We could assign an integer for each category, but it could imply relations between categories (one category is more important than the other), which do not exist.

Finally we remove the *Id* variable, which uniquely identifies each of the entries, hence giving absolutely no information to the machine learning models.

4 Methods

In this section we describe the regression algorithms used. For baseline prediction we use an mean model, which always returns the mean value of the target variable and an ordinary least squares linear regression. This way we can see if the rest of the algorithms perform better than the most basic ones. Hyperparameters for the models were found empirically or from other similar regression problems.

4.1 Regularized linear models

Regularized linear models are like linear regression, but introduce a new term to it's loss function. Ordinary least squares linear regression minimizes the loss function (residual sum of squares or RSS) defined in equation (2), where n is the number of training entries, m number of features, y_i the ground truth target variable

and $\beta_j x_{ij}$ product of the coefficient and value of it's corresponding feature.

$$RSS = \sum_{i=1}^{n} \left(y_i - \sum_{j=1}^{m} \beta_j x_{ij} \right)^2 \quad (2)$$

β_j are the weights the model is trying to learn. Problem arises when there is a lot of noise in the training data. When during training the model will try to adjust the coefficients, the estimated coefficient won't generalize well to unseen data. Model will overfit to training data, which leads to poor performance on new data - model doesn't generalize well. We introduce regularization, which penalizes the flexibility (tendency to overfit) of our model by introducing an additional term to the loss function. This term imposes a cost on high coefficients. It prevents overfitting, because the coefficients won't become unreasonably high, since this would increase the loss we are trying to minimize.

Lasso regression [6] is a linear model which introduces L1 norm of coefficients in order to penalize high coefficients. It's loss function is defined in equation (3). Parameter λ is a tuning parameter which defines how much our model will be regularized. By increasing λ we force our model to generalize more and become less flexible.

$$LASSO\_REG = RSS + \lambda \sum_{j=1}^{m} |\beta_j| \quad (3)$$

Ridge regression [7] introduces L2 norm to penalize unreasonably high coefficients. It's similar to Lasso regression, with a different penalty term. Loss function is defined in equation (4), where λ is again the tuning parameter.

$$RIDGE\_REG = RSS + \lambda \sum_{j=1}^{m} (\beta_j)^2 \quad (4)$$

In our setup we used a cross-validation estimator to find the best value of the hyperparameter λ for both of the regularization methods. It cross-validates models with different hyperparameters to find the best performing one.

4.2 Support vector machine

Support vector machines [8] (SVM) for classification work by finding a hyperplane, which maximizes the margin between the closest data points to the hyperplane of two different classes, if the data is linearly separable. In reality data usually isn't linearly separable, therefore support vector machines use soft margins and kernel trick. Soft margins allow for misclassifications, since some data points can be on the other side of the hyperplane in linearly inseparable data. It tries to balance the trade-off between finding a line that maximizes the margin and minimizes the misclassification. This trade-off is defined by the penalty term C, which is a hyperparameter of SVMs. The bigger the value of C, more penalty SVM gets when it misclassifies. Kernelt trick applies different transformations on existing features to acquire new features. If in the space of existing features we couldn't find a hyperplane that would separate the data points of different classes good enough, kernel trick allows us to do so in the space of transformed features. We can use multiple types of kernels, most commonly used is radial basis function kernel, which has a free parameter γ.

SVM for regression works very similarly to SVM for classification, only that it's goal is to find a curve, which minimizes the deviation of the points to it. Since the curve fits the data points pretty well, we don't care about the instances that are close to the curve - within the margin defined by the ϵ parameter. Instances within this margin don't incur a cost on the loss function. Instances outside the margin incur a cost in the loss function, which we want to minimize, thus SVM for regression also doesn't maximize the margin as the SVM for classification does.

In our experiments the value of C is equal to 20, ϵ to 0.008 and γ to 0.0003.

4.3 Random forest

Random forest [9] is an ensemble method. Ensemble methods combine predictions from multiple separate machine learning models together to make more accurate predictions from

any individual model. Ensemble methods for regression use multiple models that have low bias and high variance, whose predictions are averaged to acquire the final prediction. Averaging removes the high variance of the individual models, while keeping low bias.
Random forest regressor uses multiple regression decision trees, which are trained on a subset of training data. Subset is chosen through the process of bagging (bootstrap aggregation), which randomly samples the training set with replacement. This way each of the decision trees receives a sampled set of original data, which doesn't necessarily contain all of the features from the original set. Because decision trees are sensitive to data they are trained on, each tree will be different from another. This helps us train uncorrelated trees, which is essential for ensemble methods. When predicting with a trained model, each tree makes its own prediction and in the end all of them are averaged.
Hyperparameters for random forests are the following: the number of decision trees used, which we set to 1200, the maximum depth of a single decision tree, which is equal to 15, minimum samples required to split an internal node, set to 5 and minimum samples required for a node to become a leaf, which was also set to 5.

4.4 Gradient boosting models

Gradient boosting methods are ensemble methods, which were explained in chapter (4.3). Instead of training all of the weak models in parallel with bagging (like random forests), they use boosting.
Boosting [10] is a method of creating an ensemble, where we start by fitting an initial model to the data. Trained model is then evaluated on the data. Then a second model is built, which focuses on accurately predicting the cases in training data where the first model performed poorly. Next, the ensemble of first two methods is again evaluated on training data and cases where it performs poorly are found. We then again train a new model, focusing on weakness of the ensemble built so far. This way each successive model tries to correct the errors of the combined boosted ensemble of the previous models built.
Gradient boosting [11] is a type of boosting, which tries to built the best possible next model, that will minimize the overall prediction error when combined with previous models. It calculates the residuals (errors from predictions on the dataset), then tries to find patterns and train a weak classifier to correct those errors. This is done until residuals have no more patterns that could be modeled. We do this by minimizing the loss function, until the test loss reaches minimum, to prevent overfitting.
In out experiments we used two gradient boosting algorithms, named **XGBoost** [12] and **LightGBM** [13]. For XGBoost we used 6000 weak estimators with max depth of 4 and learning rate set to 0.01. For LightGBM we used 7000 estimators with learning rate set to 0.01.

5 Results

In this section we first define the metrics used for evaluation of our models, then present the results of experiments.
In experiments we used 10-fold cross validation for all of our models. Reported metrics are a mean from all of the folds of cross validation for a single model.

5.1 Metrics

In order to evaluate the performance of our models we used two metrics. First one is root mean square error (RMSE) described in equation (5). RMSE is used to give an estimation of error in same units as the *SalePrice* target variable.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2} \tag{5}$$

Second one is root mean square error between the natural logarithm of predicted *SalePrice* and natural logarithm of observed *SalePrice* (RMSLE), defined in equation (6). We use a second metric, because it estimates the error in a more correct way. Since we take a natural logarithm

of the observed and predicted *SalePrice*, the errors in predicting expensive and cheap houses will affect the result equally.

$$RMSLE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(ln(\hat{y}_i) - ln(y_i))^2} \quad (6)$$

5.2 Evaluation

Results of our experiments can be seen in table (1). The best performing model is LightGBM, which has the lowest error in RMSE and RMSLE. Ridge and Lasso regression surprisingly outperformed some powerful and more complex models like XGBoost and random forest in terms of RMSLE. We can also see the importance of the RMSLE metric. All of the models, except LightGBM, average model and linear regression, have a similar RMSE values, but differ quite a lot in RMSLE metric.

Results from evaluation of linear regression stand out. The RMSE value is missing, since the value was too big to be evaluated in Python programming language. It's also performing way worse than any other model, even the most basic average model performs way better. We can check what is happening by looking at the coefficients of the model. First we sort the coefficients from smallest to biggest, then take a look at both ends of the spectrum. The two highest coefficients are the ones corresponding to the feature *TotalBsmtSF* (total basement square feet) with value 6660131.9 and to *TotalSQR* (total square feet of the house) with value 6281510.4. On the lower end we have two coefficients corresponding to *BsmtFinSF2* (square feet of finished basement with category type 2) with value -6281510.5 and *BsmtFinSF1* (square feet of finished basement with category type 1) with the same value. If we continue looking on both ends, we see that a lot of coefficients have really big or really small values. This is a sign that the linear regression model overfit to training data and doesn't generalize well with unseen data. We can see the importance of regularization, which penalizes unreasonably big or small coefficients, since Lasso and Ridge regression perform really well. All of our algorithms also performed better than the baseline average model, which means that they actually learned useful information.

TABLE 1
Evaluation results from 10-fold cross-validation on our machine learning models. RMSE values for linear regression are missing, since the number was too big to be evaluated in Python programming language.

	RMSE	RMSLE
Average model	80183.79	0.3986
Linear regression	/	26327.0962
Ridge regression	30595.35	0.1264
Lasso regression	30041.50	0.1236
SVM	30077.79	0.1225
Random forest	30236.79	0.1435
XGBoost	30712.82	0.1403
LightGBM	**24723.86**	**0.1195**

6 Conclusion

In this work we analyze the performance of different regression models for predicting house prices. Dataset used is a collection of sales that occurred in Ames, Iowa in a period of 4 years. Each entry contains 79 features describing different characteristics of the house. We found that features that have the highest correlation to our target variable *SalePrice* are features describing the overall quality of the house, features describing the size of different parts of the house and features describing the garage. In order to fix the skewness in our features we used Box-Cox transform and natural logarithm transform. We also engineered new features that give information we believe a buyer would consider useful when purchasing a house. Two regularized models were used (Ridge and Lasso regression), a support vector machine, a random forest and two gradient boosting methods - XGBoost and LightGBM. Evaluation has shown that the best performing model is LightGBM, which achieved average RMSE value of 24723.86.

For real world application we would still have to improve the models, since the error is still to big to give an accurate estimate. For now it could be used as a rough estimate or to put a house into a certain price interval.

REFERENCES

[1] D. De Cock, "Ames, iowa: Alternative to the boston housing data as an end of semester regression project," *Journal of Statistics Education*, vol. 19, no. 3, 2011.

[2] T. A. Downes and J. E. Zabel, "The impact of school characteristics on house prices: Chicago 1987–1991," *Journal of urban economics*, vol. 52, no. 1, pp. 1–25, 2002.

[3] H. Selim, "Determinants of house prices in turkey: Hedonic regression versus artificial neural network," *Expert systems with Applications*, vol. 36, no. 2, pp. 2843–2852, 2009.

[4] J. J. Ahn, H. W. Byun, K. J. Oh, and T. Y. Kim, "Using ridge regression with genetic algorithm to enhance real estate appraisal forecasting," *Expert Systems with Applications*, vol. 39, no. 9, pp. 8369–8379, 2012.

[5] G. E. Box and D. R. Cox, "An analysis of transformations," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 26, no. 2, pp. 211–243, 1964.

[6] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 58, no. 1, pp. 267–288, 1996.

[7] P. Kennedy, *A guide to econometrics*. John Wiley & Sons, 2008.

[8] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[9] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[10] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of computer and system sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[11] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.

[12] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[13] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," in *Advances in neural information processing systems*, 2017, pp. 3146–3154.