DATA MINING AND MACHINE LEARNING

FINAL PROJECT

«House prices. Advanced regression techniques»

Student: Tatsiana Palikarpava

Contents

Introduction

This work contains an overview of using a particular machine learning model in practice based on opened Kaggle competition “House prices. Advanced regression techniques”. The idea of this competition is that we are given two datasets with 79 explanatory variables of different kind, describing various features of houses, but the first dataset contains prices of houses, while the second one does not. Thus, we are expected to predict prices for the second dataset, generate a submission and upload it to Kaggle platform, where the submission is compared with real price values and the error rate is calculated. The lower is error rate – the higher is your place in the competition.

My goal was to perform the whole process of data mining, gradually increasing accuracy by applying different approaches. I also wanted to explore the random forest regressor model and analyze some aspects of its behaviour. All the work was implemented in python with the help of scikit-learn, prince, rfpimp and other libraries. Moreover, I was trying to include illustrations to make my conclusions more visual.

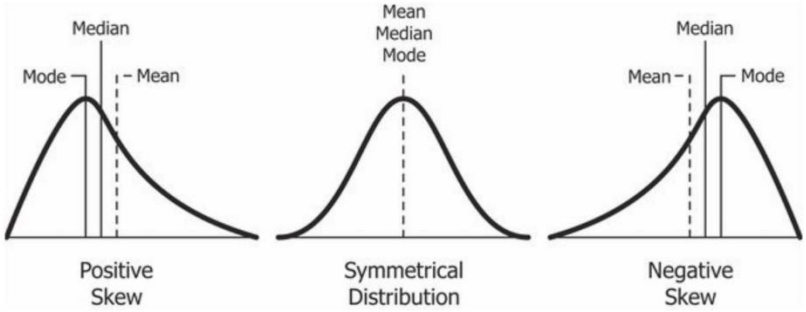
You can acquaint with all the issues you consider worth at a website of this competition, which you can find among references.

Data preprocessing

In this chapter all the steps I considered necessary before applying particular machine learning model are described. As far as the data we deal with is collected in the real world, it contains a lot of factors that can decrease the effectiveness of our model or even make it inapplicable. That is why we need to preprocess the data.

Skewness

Skewness is asymmetry in a statistical distribution, in which the curve appears distorted or skewed either to the left or to the right. Skewness can be quantified to define the extent to which a distribution differs from a normal distribution.



Regression models usually assume that the errors committed by the model have the same variance, therefore it should not make small errors for small feature values and big errors for big values. That is why such thing as skewness is not desirable and can have negative effect on the quality of the model.

One of the most popular and easy ways to reduce skewness is logarithm transformation. We simply calculate logarithm of sale prices and continue learning the model using these values, and then, in the end, when we need to generate a submission, we will calculate an exponential function of predicted “transformed” prices.

Using seaborn library we display the distribution of sale prices in training dataset.

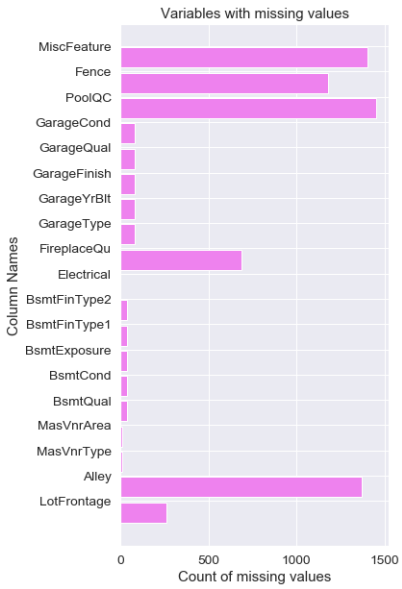
Obviously, it is positively skewed. Hence, we apply logarithm transformation and obtain the following.

Missing values

When I have just downloaded the dataset I decided to take a look at features and their values. What I have noticed is that many of them contain NA values, so I supposed that it would be useful to understand where we can face with the missed data and what we can do with this kind of problem.

Below you can take a look at some statistics concerning missing data, extracted from training dataset.

- Id 1460 non-null int64
- MSSubClass 1460 non-null int64
- MSZoning 1460 non-null object
- **LotFrontage 1201 non-null float64**
- LotArea 1460 non-null int64
- Street 1460 non-null object
- **Alley 91 non-null object**
- LotShape 1460 non-null object
- LandContour 1460 non-null object
- Utilities 1460 non-null object
- LotConfig 1460 non-null object
- LandSlope 1460 non-null object
- Neighborhood 1460 non-null object
- Condition1 1460 non-null object
- Condition2 1460 non-null object
- BldgType 1460 non-null object
- HouseStyle 1460 non-null object
- OverallQual 1460 non-null int64
- OverallCond 1460 non-null int64
- YearBuilt 1460 non-null int64
- YearRemodAdd 1460 non-null int64
- RoofStyle 1460 non-null object
- RoofMatl 1460 non-null object
- Exterior1st 1460 non-null object
- Exterior2nd 1460 non-null object
- **MasVnrType 1452 non-null object**
- **MasVnrArea 1452 non-null float64**
- ExterQual 1460 non-null object
- ExterCond 1460 non-null object
- Foundation 1460 non-null object
- **BsmtQual 1423 non-null object**
- **BsmtCond 1423 non-null object**
- **BsmtExposure 1422 non-null object**
- **BsmtFinType1 1423 non-null object**
- BsmtFinSF1 1460 non-null int64
- **BsmtFinType2 1422 non-null object**
- BsmtFinSF2 1460 non-null int64
- BsmtUnfSF 1460 non-null int64
- TotalBsmtSF 1460 non-null int64
- Heating 1460 non-null object
- HeatingQC 1460 non-null object
- CentralAir 1460 non-null object
- **Electrical 1459 non-null object**
- 1stFlrSF 1460 non-null int64
- 2ndFlrSF 1460 non-null int64
- LowQualFinSF 1460 non-null int64
- GrLivArea 1460 non-null int64
- BsmtFullBath 1460 non-null int64
- BsmtHalfBath 1460 non-null int64
- FullBath 1460 non-null int64
- HalfBath 1460 non-null int64
- BedroomAbvGr 1460 non-null int64
- KitchenAbvGr 1460 non-null int64
- KitchenQual 1460 non-null object
- TotRmsAbvGrd 1460 non-null int64
- Functional 1460 non-null object
- Fireplaces 1460 non-null int64
- **FireplaceQu 770 non-null object**
- **GarageType 1379 non-null object**
- **GarageYrBlt 1379 non-null float64**
- **GarageFinish 1379 non-null object**
- GarageCars 1460 non-null int64
- GarageArea 1460 non-null int64
- **GarageQual 1379 non-null object**
- **GarageCond 1379 non-null object**
- PavedDrive 1460 non-null object
- WoodDeckSF 1460 non-null int64
- OpenPorchSF 1460 non-null int64
- EnclosedPorch 1460 non-null int64
- 3SsnPorch 1460 non-null int64
- ScreenPorch 1460 non-null int64
- PoolArea 1460 non-null int64
- **PoolQC 7 non-null object**
- **Fence 281 non-null object**
- **MiscFeature 54 non-null object**
- MiscVal 1460 non-null int64
- MoSold 1460 non-null int64
- YrSold 1460 non-null int64
- SaleType 1460 non-null object
- SaleCondition 1460 non-null object
- SalePrice 1460 non-null int64



Python scikit-learn library provides an instrument Imputer that allows to fill in missing data using different strategies. Generally, its guesses are based on the data we already have, thus it does not make much sense in case of such features as MiscFeature, Fence, etc., because they have a very significant percentage of missing values. Therefore, the decision was the following: columns with more than 15% of missing data were eliminated, while the others were processed with sklearn.impute.**SimpleImputer.** It offers several strategies:

strategy : *string, optional (default="mean")*

The imputation strategy.

- If "mean", then replace missing values using the mean along each column. Can only be used with numeric data.
- If "median", then replace missing values using the median along each column. Can only be used with numeric data.
- If "most_frequent", then replace missing using the most frequent value along each column. Can be used with strings or numeric data.
- If "constant", then replace missing values with fill_value. Can be used with strings or numeric data.

I used 'mean' for numerical data and 'most_frequent' for categorical. There is also a more sophisticated imputer in scikit-learn called IterativeImputer, which estimates features as a function of other features of the dataset. It is experimental now and works with numerical data, but I also tried to use it (the score did not change noticeably).

One-hot encoding

In the case of random forest regressor implemented in scikit-learn categorical data is required to be converted to numerical form. The easiest way to do it is integer encoding. In this case integer numbers are assigned to the categories. Unfortunately, it can serve only if the data has natural ordering and is definitely not a reasonable solution in some cases. For example, if we have variable 'animal' with possible values 'cat', 'dog' and 'bird' and corresponding integers 1, 2 and 3, then the model can assume some ridiculous things like average of 'cat'(1) and 'bird'(3) is $\frac{1+3}{2} = 2$ ('dog') or it can produce non-integer values which is also undesirable. That is why I used one-hot encoding approach which consists in creating binary variables like 'is_cat', 'is_dog', 'is_bird', where only one of them is equal to one for each data sample.

animal	animal_num
'cat'	1
'dog'	2
'bird'	3

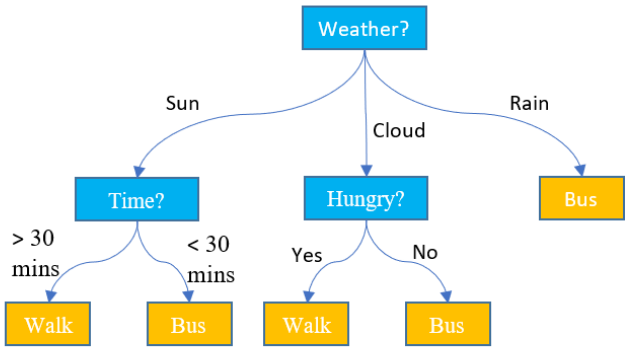
→

animal	is_cat	is_dog	is_bird
'cat'	1	0	0
'dog'	0	1	0
'bird'	0	0	1

Price prediction

The model selected for prediction and investigation was random forest regressor, which is powerful and on the other hand rather intuitive in its approach.

A building block of this model is decision tree. Basically, the decision tree splits data by considering each feature in training data.



The mean of responses of the training data inputs of particular group is considered as prediction for that group. The below function is applied to all data points and cost is calculated for all candidate splits. The split with lowest cost is chosen.

$$\sum (y_{real} - y_{predicted})^2$$

Although a tree can continue growing until all the data is split perfectly, such approach would cause enormous tree depths and as a result overfitting. Thus, the question which rises is when to stop growing a tree. One way of doing this is to set a minimum number of training inputs to use on each leaf. Another way is to set maximum depth of the model. Maximum depth refers to the length of the longest path from a root to a leaf.

As follows from its name, random forest is created by constructing a multitude of decision trees at training time and outputting mean prediction of the individual trees. An undeniable advantage of random forests is that they correct decision trees' habit of overfitting to their training set.

A few important concepts that construct a base of random forest model are bagging (bootstrap aggregating) and feature bagging. The idea of bootstrap aggregating is that the model selects n out of n samples with replacement to create a single tree that decreases variance of the model.

Feature bagging implies that a modified tree learning algorithm selects, at each candidate split in the learning process, a random subset of the features (Typically, for a problem with p features, $\sqrt{p}$ features are used in each split). The reason to do this is that if one or a few features are very strong predictors for the response variable (target output), these features will be selected in many of the trees, causing them to become correlated, leading to the decrease of model's efficiency.

Model learning

Hyperparameter tuning

To use random forest regressor we need to set its parameters. Some of them are:

- The number of trees in the forest.
- The function to measure the quality of a split.
- The maximum depth of the tree.
- The minimum number of samples required to split an internal node.
- The minimum number of samples required to be at a leaf node.
- The number of features to consider when looking for the best split.

These parameters are not amended during learning, but directly assigned by a programmer, that is why they are called hyperparameters. The problem is that in the majority of the cases we do not know which values of hyperparameters are optimal.

The process of hyperparameter tuning (also called hyperparameter optimization) means finding the combination of hyperparameter values for a machine learning model that performs the best – as measured on a validation dataset – for a problem. Scikit-learn offers two instruments for hyperparameter tuning.

sklearn.model_selection.**GridSearchCV:**

> Set up a grid of hyperparameter values and for each combination, train a model and score on the validation data. In this approach, every single combination of hyperparameters values is tried which can be very inefficient.

sklearn.model_selection.**RandomizedSearchCV:**

> Set up a grid of hyperparameter values and select *random* combinations to train the model and score. The number of search iterations is set based on time/resources.

Generally, it makes sense to apply grid search only for small amount of parameters to try, thus I used randomized search. The sources I have explored say that randomized search is likely to find although not the best, but close to the best set of parameters.

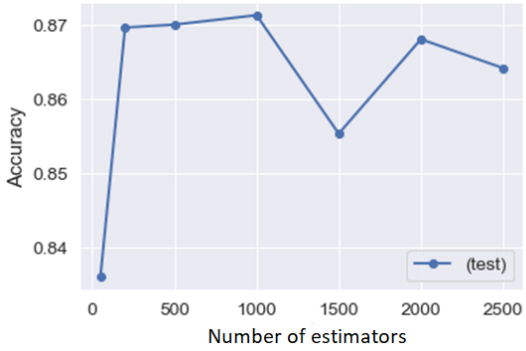
Performance analysis

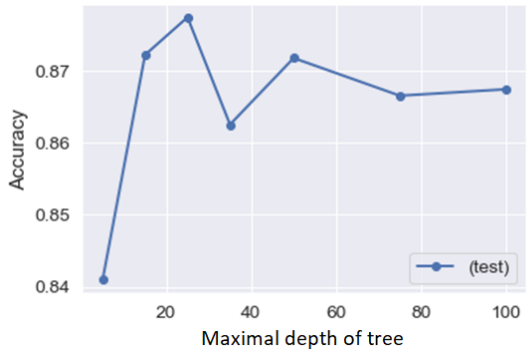
I applied sklearn.model_selection.**RandomizedSearchCV** (100 iterations, 5-fold cross validation) with the following grid:

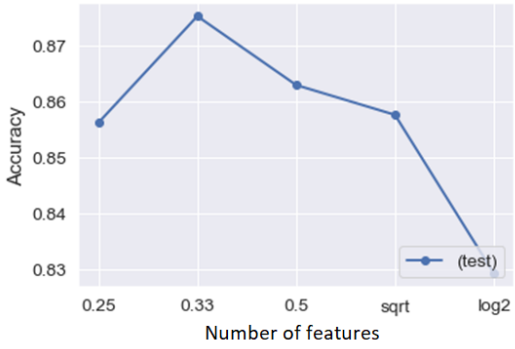
Number of trees in random forest: [50, 200, 500, 1000, 1500, 2000, 2500]
Number of features to consider at every split: [0.2, 0.33, 0.5, 'sqrt', 'log2']
Maximum number of levels in the tree: [5, 15, 25, 35, 50, 75, 100]
Minimum number of samples required to split a node: [2, 4, 6, 8]
Minimum number of samples required at each leaf node: [1, 2, 5, 10]

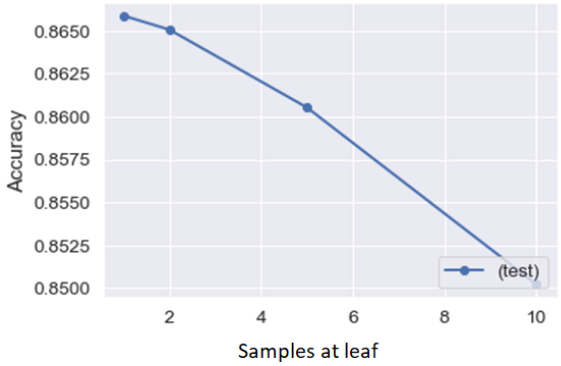
The result was:
Number of trees in random forest: 1000
Number of features to consider at every split: 0.33
Maximum number of levels in the tree: 25
Minimum number of samples required to split a node: 2
Minimum number of samples required at each leaf node: 1

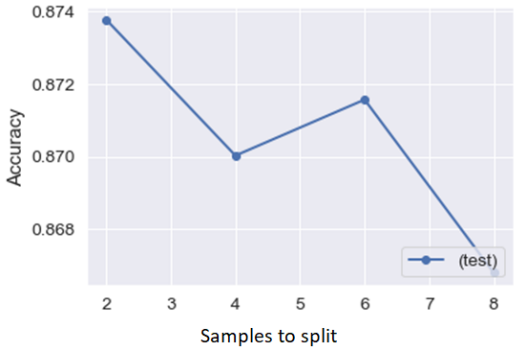
The next step was to look at the behaviour of the model in case of changing the parameters chosen as optimal. I split the training data using train_test_split (test_size = 0.3) and assessed the average accuracy in 5 rounds. Below you can see several plots that show how the change of these parameters affected model's accuracy.











There exist several issues to keep in mind. The first is that the accuracy obtained in train_test_split is a little bit higher than the real one shown by Kaggle platform. It is also necessary to remember that we applied RandomizedSearch, so we are not guaranteed that this set is optimal.

From what I can gather, the most significantly on the final prediction influence number of estimators, number of features and tree depth.

When I obtained these nearly optimal parameters, I considered curious to see whether they are so good from the point of view of Kaggle by generating submissions with values of parameters lower and higher than 'optimal' ones.

According to the plot, the accuracy tends to rise with the increase of number of estimators until 1000 and then starts to reduce. Nevertheless, when I compared this result to Kaggle response, I was surprised to notice that the best accuracy was obtained with 1500-1600 estimators, so the maximum point is a bit displaced. However, the other results appeared to be rather plausible and changes in their values were decreasing the score.

As I mentioned earlier, very often for a problem with p features, $\sqrt{p}$ features are used in each split, which is not the case here. I have read that this value is more wide-spread in classification problems, while for regression problems $p/3$ is normally used.

In the attached archive "trees.zip" you can see files "tree5.png", "tree15.png" and "tree25.png" which show the trees generated for different depth values. I do not insert them as pictures in this document, because they are too heavy and not very convenient to see in such format. It can seem strange that in many cases we have only one sample at leaf node, because such model is likely to be overfitted, but the crossvalidation estimates and the results obtained from Kaggle still show that such model behaves better than simpler one.

Post-learning analysis and dimensionality reduction

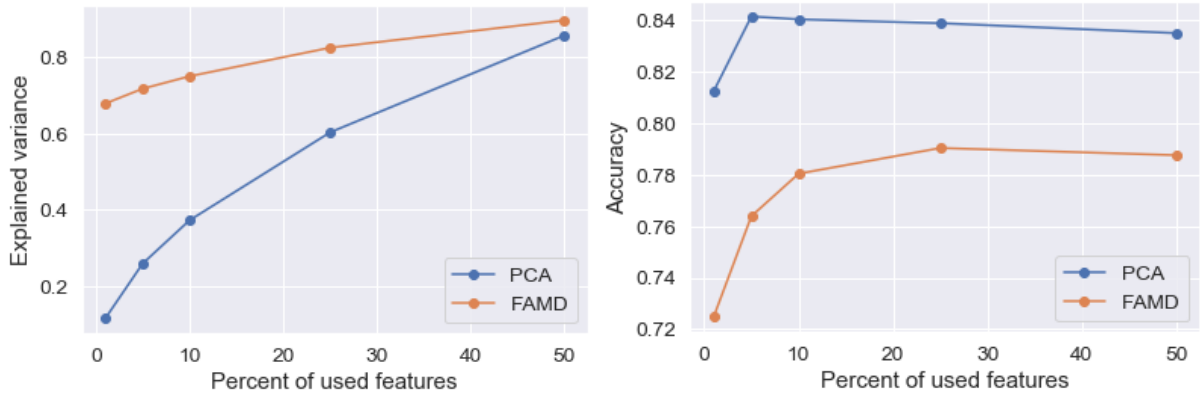
Dimensionality reduction

In this problem we have both numerical and categorical data that complicates the usage of dimensionality reduction techniques.

What I found out is that for such mixed data we can apply PCA after one-hot encoding, but it becomes less meaningful than PCA for usual numerical data and FAMD (Factor analysis of mixed data). They are implemented in the library prince in python.

Basically, I have tried these two methods with different numbers of components and that is what I have got:

		1% of features	5% of features	10% of features	25% of features	50% of features
		3 features	**15 features**	**30 features**	**74 features**	**148 features**
PCA	**Explained variance**	0.1144005	0.25866916	0.37200645	0.60260517	0.85668277
	Accuracy	0.81244	0.84154	0.84042	0.83892	0.83505
		2 features	**4 features**	**7 features**	**18 features**	**37 features**
FAMD	**Explained variance**	0.67804529	0.71699702	0.74958445	0.82446322	0.89637598
	Accuracy	0.72473	0.76392	0.78043	0.79041	0.78761

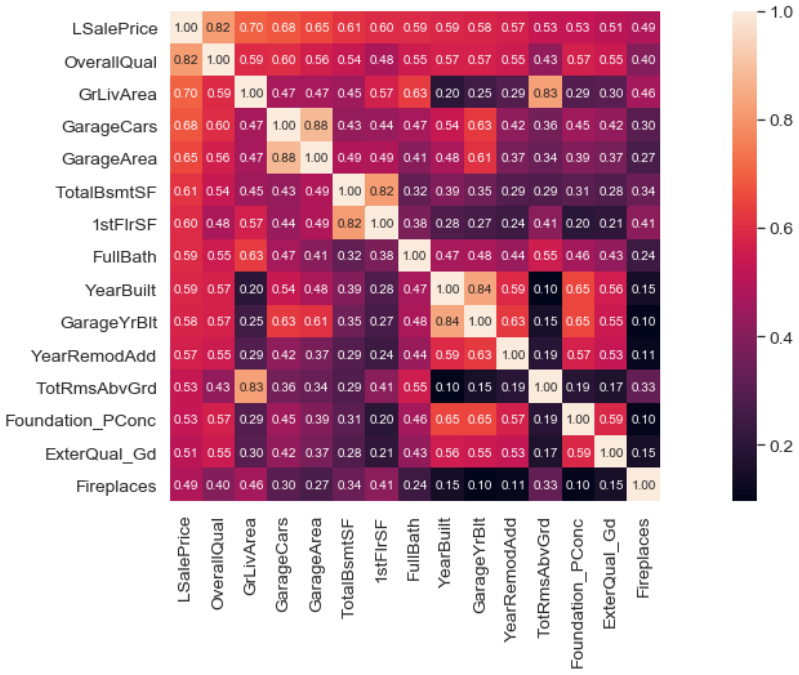


In this case we can see that PCA let us obtain better score than with FAMD, but anyway, we can see that even a small percent of features allows to make predictions of ~80% accuracy, while the model becomes much easier.

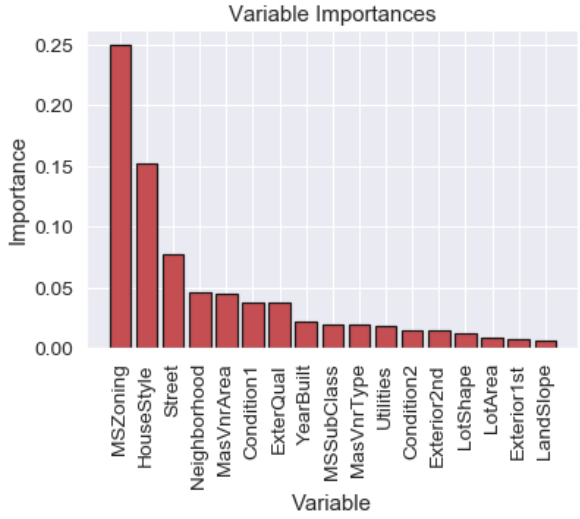
Feature importance

In this chapter I would like to analyze which features are more likely to affect the final prediction and whether we can simplify the model by eliminating those of them that do not contribute much.

The first guess was that the features that are the most correlated with ´LSalePrice´ value would be the most important in price prediction. I have constructed a heatmap of 15 most correlated with ´LSalePrice´ features:



Random forest regressor allows to obtain feature importance that is calculated based on how much each feature contributes on average to decreasing the weighted impurity. Below you can see features with the highest default feature importance values.



So then, we can compare these two lists of features. Surprisingly, they have almost nothing in common.

OverallQual	MSZoning
GrLivArea	HouseStyle
GarageCars	Street
GarageArea	Neighborhood
TotalBsmtSF	MasVnrArea
1stFlrSF	Condition1
FullBath	ExterQual
YearBuilt	**YearBuilt**

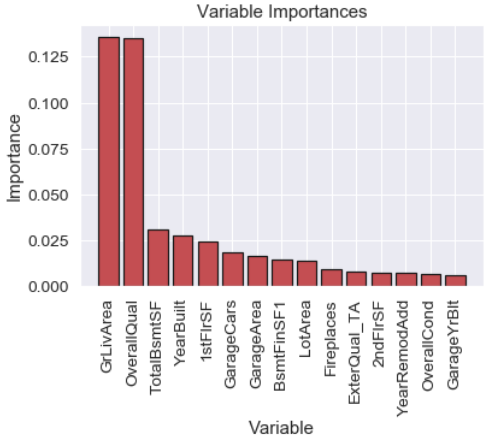
GarageYrBlt	MsSubClass
YearRemodAdd	MasVnrType
TotRmsAbvGrd	Utilities
Foundation_Pconc	Condition2
ExterQual_Gd	Exterior2nd
Fireplaces	LotShape

Unfortunately, this easy approach has several serious drawbacks. Firstly, feature selection based on impurity reduction is biased towards preferring variables with more categories. Secondly, when the dataset has two (or more) correlated features, then from the point of view of the model, any of these correlated features can be used as the predictor, with no concrete preference of one over the others. But once one of them is used, the importance of others is significantly reduced since effectively the impurity they can remove is already removed by the first feature. As a consequence, they will have a lower reported importance. When interpreting the data, it can lead to the incorrect conclusion that one of the variables is a strong predictor while the others in the same group are unimportant, while actually they are very close in terms of their relationship with the response variable.

Another way to estimate feature importance is calculate permutation feature importance. This approach directly measures feature importance by observing how random reshuffling (thus preserving the distribution of the variable) of each predictor influences model performance.

The approach can be described in the following steps:

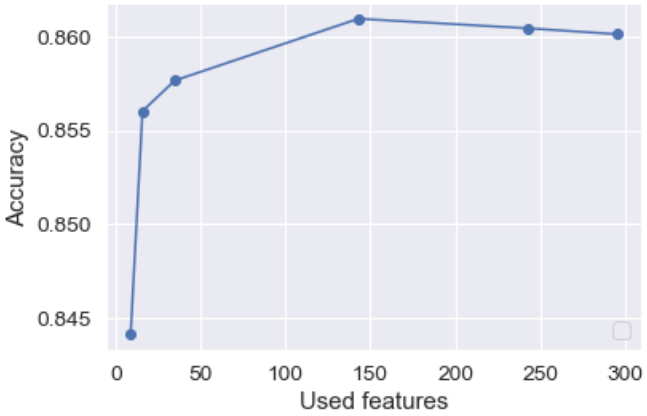
- Train the baseline model and record the score (accuracy/R^2/any metric of importance) by passing the validation set (or OOB set in case of Random Forest). This can also be done on the training set, at the cost of sacrificing information about generalization.
- Reshuffle values from one feature in the selected dataset, pass the dataset to the model again to obtain predictions and calculate the metric for this modified dataset. The feature importance is the difference between the first score and the one from the modified (permuted) dataset. Repeat this for all features in the dataset.



OverallQual	**GrLivArea**
GrLivArea	**OverallQual**
GarageCars	**TotalBsmtSF**
GarageArea	**YearBuilt**
TotalBsmtSF	**1stFlrSF**
1stFlrSF	**GarageCars**
FullBath	**GarageArea**
YearBuilt	BsmtFinSF1
GarageYrBlt	LotArea
YearRemodAdd	**Fireplaces**
TotRmsAbvGrd	ExterQual_TA
Foundation_Pconc	**YearRemodAdd**
ExterQual_Gd	OverallCond
Fireplaces	**GarageYrBlt**

Now the result is much more optimistic: many of correlated with the final result features significantly contribute to the final prediction. Based on this feature importance calculation I selected a set of features with very low importance and did not use them to learn the model and it slightly improved the score.

	Importance threshold: 0.01	**Importance threshold: 0.005**	**Importance threshold: 0.001**	**Importance threshold: 0.0001**	**Importance threshold: 0.00001**	**No features eliminated**
Features eliminated	286 (9 used)	279 (16 used)	260 (35 used)	152 (143 used)	52 (243 used)	0 (295 used)
Accuracy	0.84411	0.85602	0.85765	0.86094	0.86043	0.86012



Conclusion

All in all, my work consists of the following investigation:

- Data preprocessing (reducing skewness, imputing missing values, one-hot encoding)
- Training Random Forest Regressor
- Hyperparameters tuning
- Analysis of the influence of hyperparameters on the result
- Attempt of dimensionality reduction
- Feature importance analysis

After all this steps my model is able to predict house price with accuracy of 86,094%.

References

1. https://towardsdatascience.com/random-forest-in-python-24d0893d51c0
2. https://www.kaggle.com/c/house-prices-advanced-regression-techniques/overview
3. https://medium.com/datadriveninvestor/random-forest-regression-9871bc9a25eb
4. https://www.kaggle.com/pmarcelino/comprehensive-data-exploration-with-python
5. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html
6. https://www.kaggle.com/willkoehrsen/intro-to-model-tuning-grid-and-random-search
7. https://blog.datadive.net/selecting-good-features-part-iii-random-forests/
8. https://towardsdatascience.com/explaining-feature-importance-by-example-of-a-random-forest-d9166011959e
9. https://www.analyticsvidhya.com/blog/2018/08/dimensionality-reduction-techniques-python/
10. https://github.com/MaxHalford/prince