House Prices: Advanced Regression Techniques
Tópicos Especiais em Sistemas de Controle
Universidade Federal do Rio de Janeiro – UFRJ

Por Lucas Barcellos Oliveira

## Introdução

Este relatório aborda as abordagens e decisões tomadas para alcançar um modelo de predição para o problema House Prices: Advanced Regression Techniques, disponível como competição[1] na plataforma online Kaggle. Tal trabalho é parte dos critérios de avaliação da disciplina Tópicos Especiais em Sistemas de Controle, da Universidade Federal do Rio de Janeiro, lecionada pelo Prof. Heraldo L. S. Almeida no semestre 2018.1. A disciplina em questão discorre sobre o tema de Aprendizado de Máquina.

Serão discutidas as conclusões obtidas a partir de uma análise exploratória dos dados providos, bem como as famílias de modelos testadas e seus desempenhos. Dessa forma, espera-se proporcionar um bom entendimento sobre o problema em si e acerca de métodos para "resolvê-lo", isto é, predizer novas observações com erro mínimo.

O problema abordado, House Prices: Advanced Regression Techniques, é um problema de regressão que oferece atributos, numéricos e categóricos, sobre aparência, características e estado de imóveis em Ames (Iowa, EUA), atrelados ao preço pelo qual foram vendidos em seu conjunto de treino. Visa-se, portanto, determinar uma maneira pela qual podemos descrever tal valor para imóveis, no conjunto de testes, apenas a partir de tais informações.

Foi utilizada a interface de desenvolvimento Spyder dentro do software Anaconda na elaboração e execução dos scripts que viabilizaram este trabalho. O pacote Scikit-learn para a linguagem de programação Python foi utilizada para uma manipulação veloz e prática dos diversos modelos de treinamento. Ademais, foram utilizadas mais alguns pacotes para Python para representação, visualização e transformação dos dados, como Pandas, Numpy e Seaborn, respectivamente.

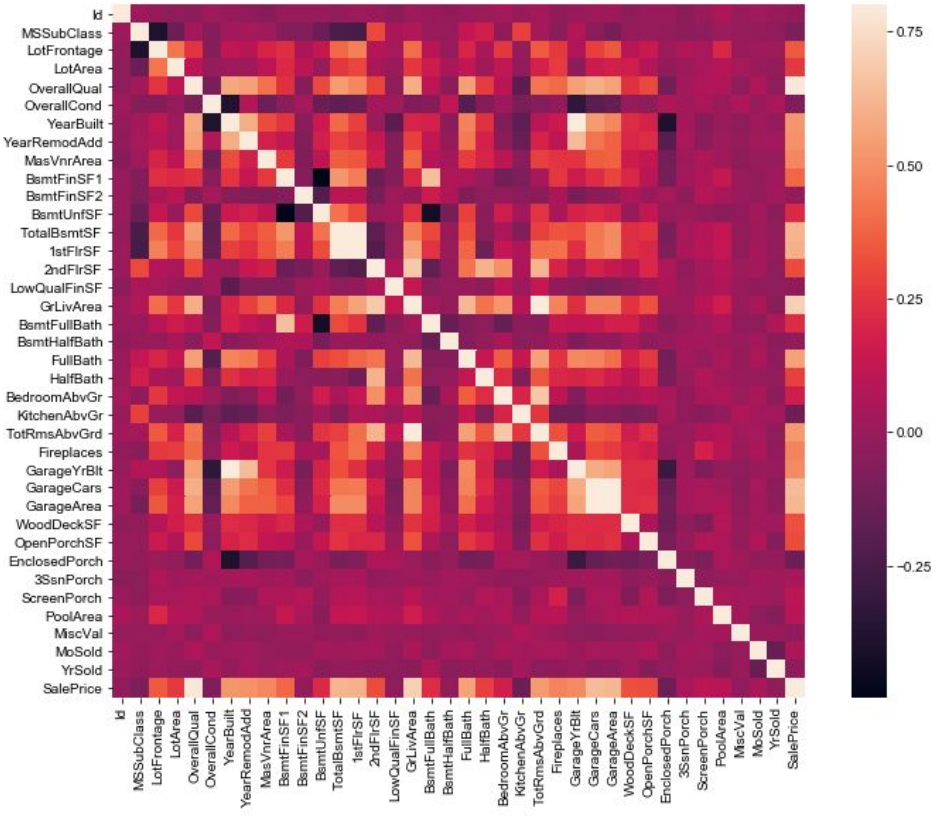
## Análise exploratória

O primeiro passo para enfrentar esse desafio é conhecer bem o problema. Isso nos permite avaliar que modelos são mais indicados – provavelmente apresentando melhor capacidade de predição – e nos auxilia em momentos nos quais o desempenho pode deixar a desejar, mostrando como aprimorá-lo.

O primeiro passo tomado foi estudar algumas estatísticas básicas relacionadas ao conjunto de treinamento do problema. Utilizando o método .describe() para cada atributo obtemos informações como valor médio, máximo e moda para variáveis numéricas. Por exemplo, para o atributo “SalePrice” que desejamos prever, vemos:

```
count      1460.000000
mean     180921.195890
std       79442.502883
min       34900.000000
25%      129975.000000
50%      163000.000000
75%      214000.000000
max      755000.000000
Name: SalePrice, dtype: float64
```

Uma das melhores formas de atingir esse objetivo é preparar uma visualização dos dados. Por isso, foi feita, no arquivo destinado a visualização “vis.py”, a plotagem da matriz de correlação, dispersão e histogramas.




**Matriz de correlação plotada utilizando o heatmap (mapa de calor) do Seaborn**


**Matriz de projeção, que inclui dispersão e histograma para cada variável numérica**

As correlações mais altas de SalePrice são com OverallQual (0.790982) e GrLivArea (0.708624). Logo, podemos esperar que ambas as variáveis tenham grande peso no modelo final.

## Pré-processamento

Em primeiro lugar, é eliminada a coluna “Id” do conjunto de dados visto que o modelo não deve utilizar o identificador de uma observação para a predição, apenas suas características. Isso evita que o treinamento se aproveite de alguma disposição específica das observações no conjunto de dados para prever, erroneamente, o alvo em função do Id.

Em seguida, após alguns testes, percebeu-se que remover algumas observações muito discrepantes permitia um aumento significativo do desempenho. Buscando nas informações levantadas pela análise dos dados, optou-se por remover todas as observações com atributo GrLivArea maior que 3500 utilizando:

**train.drop(train[train[“GrLivArea”]>3500].index,inplace=True)**

Outros exemplos destoantes dos demais foram armazenados em uma lista e são removidos antes do treinamento. Essa iniciativa também colabora na prevenção de overfitting.

Depois, foi feito o tratamento das variáveis categóricas. Como na grande maioria delas não era possível estabelecer ordenação entre os valores, foi utilizado o método .get_dummies() no dataframe Pandas para binarizar todas as variáveis categóricas. Tal método dispensa o uso do OneHotEncoder após LabelEncoder conforme especificado pelo Scikit-learn.

Como podemos ver na matriz de correlação, alguns atributos estão fortemente correlacionados com outros, como GarageArea e GarageCars. Nesses casos, deseja-se manter apenas uma das variáveis correlacionadas e descartar as demais, melhorando o condicionamento da matriz e evitando problemas de convergência em etapas de otimização. Para isso, foram eliminados os atributos com skew maior que 75% com outro atributo.

Passamos, então, a tratar com valores ausentes, que são marcados como nan no dataframe. Foi considerado remover os exemplos com valores ausentes, porém, visto que o percentual de observações com pelo menos um valor ausente era considerável, realizar tal descarte impactava negativamente no desempenho dos modelos.

Por isso, foi escolhido preencher os valores faltantes. Foram testados preenchimentos com zero ou com a média das demais observações para aquele atributo, obtendo desempenhos idênticos. Foi decidido, então, utilizar o preenchimento com zeros, dado seu significado. Em variáveis como “Alley”, por exemplo, o valor nan representa que não existe uma viela para aquele imóvel.

Por fim, foi realizada a normalização dos valores do conjunto de dados. Após testar as três formas de normalização sugeridas (MaxAbsScaler, MinMaxScaler e StandardScaler), determinou-se que MinMaxScaler e StandardScaler proporcionaram os melhores desempenhos, dependendo do tipo de modelo utilizado. Assim, a normalização mais adequada é utilizada dependendo do teste.

Buscou-se também utilizar o PolynomialFeatures para aumentar a dimensão do espaço de atributos. No entanto, não houve ganho significativo levando-o ao grau 2. Contrariamente, foi percebida grande demora no treinamento em virtude do número de variáveis. Não foi possível testar elevar as variáveis ao grau 3 por ultrapassar o limite de memória RAM disponível no computador utilizado.

## Modelos

Para guiar o caminho em meio ao vasto universo de modelos disponíveis no Scikit-learn, sempre foi consultada, para cada família de modelos, a lista com todas as suas variações implementadas no pacote.[2]

Em todos os testes, foi realizada validação cruzada com 10 iterações, ou seja, dividindo o conjunto de treino em 90% de observações para treinamento e 10% para validação a cada passo. Para tal, foi utilizado cross_val_score do módulo sklearn.model_selection, que realiza a validação cruzada e calcula o desempenho do modelo por si só.

## Vizinhos mais próximos

Optou-se por iniciar com uma família de modelos com funcionamento mais simples, mas que pode resultar em desempenho aceitável dependendo do problema estudado: o k-NN.

Para conhecer o número adequado de vizinhos para o modelo, foi utilizado um loop for que treina sucessivos modelos variando o parâmetro n_neighbors do construtor de KNeighborsRegressor.

```
#%% kNN - Peso Uniforme - Distância Euclidiana
for i in range(1,21):
    model=neighbors.KNeighborsRegressor(n_neighbors=i,p=1)

    scores=cross_val_score(model,X,Y,cv=10)

    print("\n"+str(i)+"-kNN - R2:")
    print(scores.mean())
    print(str(i)+"-kNN - Desvio Padrão:")
    print(scores.std())
```

**Loop for responsável por mostrar desempenho de k-NN para diversos valores de k**

Dessa maneira, percebeu-se que o número de vizinhos que maximiza o desempenho do modelo é k=8 utilizando os parâmetros default de peso uniforme e distância euclidiana. Tal modelo resultou em R2 médio de 0.72176 e desvio padrão de 0.05882.

Em seguida, vasculhando os possíveis parâmetros de entrada para treinamento de modelos k-NN no Scikit-learn, foi testado atribuir ao parâmetro o valor “distance”, dando pesos ponderados a cada vizinho inversamente proporcionais a suas distâncias à observação analisada. Os resultados foram um R2 médio de 0.724672 e desvio padrão de 0.05878. Apesar da média ligeiramente superior, a melhoria ainda não é considerável devido ao desvio padrão. O valor ótimo de vizinhos também foi k=8 após novo teste em loop.

Foi testada, então, a variação utilizando distância de Manhattan no lugar da distância euclidiana padrão. Para o atributo p=1 foi passado ao construtor. Foram feitos testes para o peso uniforme entre vizinhos e para peso ponderado em função da distância, assim como nos casos anteriores. Para cada caso, também foram utilizados loops para determinar o melhor número de vizinhos. Neste caso, o número que resultou em melhor desempenho em ambos os casos foi k=6.

Para distância de Manhattan e peso uniforme foi obtido R2 médio de 0.74741 com desvio padrão de 0.05600. Já para a mesma métrica de distância mas peso ponderado em função da distância, temos R2 médio de 0.75072 e desvio padrão de 0.05584, representando o melhor desempenho obtido para modelos de vizinhos mais próximos.

## Modelos Lineares

Foram testados, em seguida, modelos um pouco mais sofisticados. A primeira classe de modelos a se estudar, então, é a de modelos lineares.

Primeiramente, experimentamos o uso de um regressor com regularização Lasso. Foi utilizado o modelo LassoCV, que realiza sua própria validação para otimizar boa parte dos hiperparâmetros. A princípio, o desempenho do modelo girava em torno de um R2 de 0.8 na média. No entanto, após muito ajustar o hiperparâmetro alfa da taxa de aprendizagem para 56.0, chegamos a um R2 médio de 0.91384 e desvio padrão de 0.00999.

Naturalmente, o próximo passo foi testar o desempenho do modelo linear com regularização Ridge. Utiliza-se o regressor RidgeCV, análogo ao anterior. Dessa vez, já buscando o melhor hiperparâmetro alfa, obtemos o valor de 15.0. O desempenho do modelo alcança média na validação cruzada de R2 igual 0.91335 e desvio padrão igual a 0.00900.

Posteriormente, observando a lista dos regressores implementados pelo Scikit-learn, vemos mais três tipos de modelos promissores em nosso cenário: BayesianRidge, Huber e RANSAC.

O modelo BayesianRidge conta com uma variação do Bayesiano para efetuar o ajuste automático dos hiperparâmetros. Essa abordagem mista combinando duas classes de modelos traz excelentes resultados, com R2 médio de 0.91378 e desvio padrão de 0.00861. Por outro lado, ele é bastante sensível a mudanças feitas na etapa de pré-processamento.

O regressor de Huber se propõe a ser uma espécie de regressor bastante tolerante a valores discrepantes. Para isso, ele busca w e alfa que minimizem o erro absoluto e o erro quadrático dados por |(y - X'w) /sigma| e |(y - X'w) / sigma|^2. O desempenho, no entanto, é consideravelmente inferior, desempenhando R2 médio de 0.75130 com desvio padrão de 0.03982 na validação cruzada.

Já RANSAC é a abreviação de RANdom SAmple Consensus (Consenso de amostra randômico) e baseia-se em um algoritmo iterativo para otimizar hiperparâmetros, além de também buscar ser tolerante a outliers. No entanto, tal otimização interna segue um esquema de votação, aplicando as várias combinações de hiperparâmetros sobre um conjunto randômico de observações.

O algoritmo busca construir um conjunto de observações sem outliers reaplicando os modelos treinados em cada iteração sobre observações e considerando-as outliers caso não consigam ser bem previstas. O desempenho do modelo RANSAC foi de R2 médio de 0.82519 com desvio padrão de 0.09523.

## Ensembles

Os ensembles combinam múltiplas instâncias de preditores a fim de obter um desempenho superior ao que seria obtido com o uso de apenas uma instância individualmente.

O regressor potencializado por gradiente (gradient-boosted) é um modelo aditivo de árvores de decisão. Após a construção de uma árvore, uma nova árvore é criada de modo a melhor classificar justamente os exemplos que a árvore anterior falhou em predizer. Cada nova árvore é instanciadas de modo a minimizar o resíduo da anterior.

Um dos parâmetros de entrada é o número de estimadores que deram combinados no ensemble. Após realizar múltiplos testes variando tal valor dentro de um loop for, chegou-se ao ótimo de 110 estimadores. Esse número é passado no construtor da classe GradientBoostedRegressor com n_estimators=110. Foi alcançada média de R2 igual a 0.90884 e desvio padrão de 0.01493.

Outro célebre ensemble é o de Random Forest. Cada árvore componente desse modelo é treinada sobre um subconjunto aleatório do conjunto de observações original. Sua saída é dada por uma função das saídas de cada árvore é seu funcionamento é excelente alternativa no combate a overfitting.

A construção do ensemble, assim como no modelo anterior, se deu com o número de estimadores igual a 110. O desempenho obtido com a floresta randômica foi de R2 médio de 0.88576 e desvio padrão de 0.02037.

O ensemble AdaBoost, ou boosting adaptativo, é muito robusto a overfitting, mas ainda é suscetível a outliers. Tem grande semelhança ao gradient-boosted tree, porém, a cada iteração do treinamento, os pesos de cada exemplo são ajustados em função do erro. Seu desempenho apresentou média de R2 igual a 0.83350 e desvio padrão igual a . Após atribuir 110 ao número de estimadores, obtém-se R2 médio de 0.83455 e desvio padrão de 0.02121.

Ensembles do tipo bagging consistem em agregar árvores de decisão de modo a reduzir a variância do preditor sem elevar seu viés. A grande vantagem dessa alternativa seria a simplicidade de seu funcionamento, reduzindo o consumo de recursos computacionais. Foi alcancado 0.86877 de R2 médio e 0.02156 para seu desvio padrão. Fixando o número de estimadores em 110, temos R2 médio de 0.88408 com desvio padrão de 0.2139.

Por fim, o ensemble ExtraTrees tem seu nome vindo de “extremely randomized trees”, ou árvores extremamente randomizadas. Ele possui muitas semelhanças com o Bagging ou RandomForest, porém com decisões no interior de cada árvore também randômicas. Após ajustes, seu desempenho foi de R2 médio igual a 0.87076 com desvio padrão de 0.01578. Enfim, considerando 110 estimadores, encontramos R2 médio igual a 0.88974 e desvio padrão igual a 0.01503.

## Máquinas de vetor de suporte

Máquinas de vetor de suporte (de sigla em inglês, SVM - Support Vector Machine) constituem uma família de modelos muito robusta e que costumam apresentar notavelmente elevado desempenho em problemas considerados difíceis.

Elas mapeiam os exemplos de seu espaço original para um espaço de características em que o problema pode ser mais facilmente resolvido. Um hiperplano é utilizado para resolver o problema em tal espaço e depois mapeado de volta para o espaço original. A distância da observação até o plano é usada para determinar o valor da predição.

O hiperparâmetro de maior destaque a ser ajustado no modelo SVR (SVM para problemas de regressão) é o valor de C, o custo atribuído a uma unidade de distância do hiperplano. Utilizando a mesma metodologia de variar seu valor de troca de um intervalo de valores promissores, determinou-se que o melhor valor para C é 1350000.

Também foi crucial ajustar o valor de epsilon a distância máxima do plano que delimita uma área em que penalidades (dadas por C) não serão aplicadas. Tal hiperparâmetro age como uma forma de limiar de tolerância para erros.

Assim, utilizando o kernel padrão de funções de base radial (rbf), foi obtido desempenho de R2 médio de 0.92077 com desvio padrão de 0.00828. Removendo a deleção de possíveis outliers, atingimos média de R2 de 0.92947 e desvio padrão de 0.01079.

Foram feitos testes com variações do SVR, como NuSVR e LinearSVR. O modelo NuSVR, que utiliza uma parametrização diferente do modelo clássico, teve seu melhor desempenho com parâmetro “nu” igual a 0.74, encontrado também por uma busca implementada com um loop em Python. Seu desempenho foi de R2 médio de 0.92079 e desvio padrão de 0.00831 na validação cruzada. Desse modo, podemos considerá-lo “tecnicamente empatado” com o modelo anterior. Sem a remoção de outliers, temos 0.92956 e 0.01066, respectivamente, como R2 médio e desvio padrão. Isso representa o melhor desempenho obtido até o momento.

Finalmente, o treinamento com LinearSVR utiliza kernel linear. Apesar da relativa facilidade de cálculo, ela perde capacidade de predição para problemas mais complexos. Como era esperado, seu desempenho foi o menor entre as SVMs estudadas atingindo 0.86697 de R2 médio e 0.04444 para seu desvio padrão.

## Árvore de decisão

Posteriormente, a guisa de curiosidade, foram feitos testes utilizando um modelo de árvore de decisão convencional. Assim, temos podemos fazer uma comparação com o que verificamos nos modelos ensemble.

De fato, como era esperado, um modelo, instanciado da classe DecisionTreeRegressor do Scikit-learn, que envolve apenas uma árvore teve desempenho visivelmente pior que o dos ensembles. Foram aferidos R2 médio de 0.77357 e desvio padrão de 0.05990.

A grande vantagem dessa família de modelos é sua característica de naturalmente selecionar os parâmetros mais “importantes” na regressão e desconsiderar outros atributos altamente correlacionados. Isso, inclusive, dispensa a remoção de de variáveis correlacionadas no pré-processamento.

## XGBoost

Consultando as recomendações em fóruns especializados e as ideias implementadas por outros usuários do Kaggle, pode-se perceber que muitos utilizam uma biblioteca chamada XGBoost com resultados incrivelmente bons.

Essa biblioteca é abreviação para Extreme Gradient Boost e implementa um ensemble de árvores de decisão aprimoradas por gradiente com grande destaque para a otimização feita. Desenvolvido como software a aberto por Tianqi Chen do Distributed (Deep) Machine Learning Community, XGBoost está disponível em várias linguagens e tem sido amplamente utilizada pela comunidade de mineração de dados e aprendizado de máquina.

Ocorreram algumas dificuldades na adição do pacote no ambiente Anaconda, o que levou a buscar outras formas de instalação.[3] Foi utilizado um Python Egg não oficial disponibilizado por um grupo de pesquisa em um repositório.[4] Sua extração foi feita com pip install no terminal do Anaconda para Windows.

Existe uma API da XGBoost para Scikit-learn que permite seu uso como um modelo nativo, com a classe XGBRegressor. Foram estudados os hiperparâmetros envolvidos e passados em seu construtor, segundo a documentação[5], e os dois de maior influência são o número de árvores n_estimators e a profundidade máxima de cada instância max_depth.

O XGBoost com 650 estimadores e profundidade máxima de 3 níveis apresenta R2 médio de 0.92353 e desvio padrão de 0.01322 na validação cruzada. Já sua versão com 1100 estimadores e profundidade máxima 2 desempenha R2 médio de 0.93071 com desvio padrão de 0.01064.

Foram feitos experimentos com submodelos dart, que apresentou o mesmo desempenho que a primeira tentativa (R2 médio: 0.92353; desvio padrão: 0.01322), e linear, com desempenho muito pior de média de R2 0.55134 e desvio padrão 0.03226.

## Meu ensemble

Em um último conjunto de tentativas, visto que os modelos mais robustos e sofisticados são ensembles, buscou-se criar um ensemble próprio. Deseja-se, com isso, combinar as saídas dos melhores modelos produzidos até então para produzir uma saída mais robusta e com menos variância.

Para concretizar tal intenção, foram estudadas as formas de criar nosso próprio regressor. Para isso, a classe que representa nosso regressor deve herdar das classes BaseEstimator e RegressorMixin do Scikit-learn. Além disso, para poder ser utilizado na validação cruzada, o preditor deve possuir o método fit, que recebe como parâmetros as características e a variável alvo, e o método predict, que recebe as características dos exemplos a serem preditos.

Abaixo, a implementação feita para nosso ensemble próprio. No treino, cada modelo-base é treinado individualmente. Já na predição, é feita a média aritmética das saídas de cada um dos vários modelos.

```
from sklearn.base import BaseEstimator, RegressorMixin
class MeuEnsemble(BaseEstimator,RegressorMixin):
    def __init__(self,models):
        self.models=models
    def fit(self,X,Y):
        for i in self.models:
            print ("Ajustando modelos")
            i.fit(X,Y)
        return
    def predict(self,Y):
        pred=[]
        for i in Y:
            temp=0
            for j in self.models:
                temp+=j.predict(i.reshape(1,-1))
            temp/=len(models)
            pred+=[temp]
        return pred

model=MeuEnsemble(models)
```

**Trecho do código que implementação ensemble personalizado**

Os modelos-base utilizados são aqueles com melhor desempenho nos testes individuais. São eles os ensembles convencionais embutidos com o Scikit-learn, além de regressões com regularização Lasso e Ridge, XGBRegressor e máquinas de vetor de suporte já discutidos anteriormente. A única exceção dos demais foi o XGBRegressor com 650 árvores estimadoras, que apresentou melhor resultado em ensemble apesar de ter pior desempenho individualmente.

A implementação foi pensada de maneira a ser genérica o bastante que a simples alteração na lista “models” é o suficiente para alterar a estrutura do ensemble. Com essa abordagem, pretende-se reduzir a variância dos modelos e corrigir falhas específicas que alguns modelos possam vir a apresentar em certas observações.

```
models=[
    RandomForestRegressor(n_estimators=110),
    ExtraTreesRegressor(n_estimators=110),
    GradientBoostingRegressor(n_estimators=110),
    LassoCV(alphas=[56.0],fit_intercept=False,max_iter=1000000),
    RidgeCV(alphas=[15.0],fit_intercept=False),
    XGBRegressor(n_estimators=650),
    #KNeighborsRegressor(n_neighbors=6,weights="distance",p=1),
    SVR(C=1350000,epsilon=1200),
    NuSVR(C=1350000,nu=0.74)
]
```

**Lista de modelos utilizada no ensemble próprio**

O desempenho do ensemble foi excelente, alcançando 0.92863 de R médio e 0.01248 de desvio padrão. Foram feitos ajustes no pré-processamento, mudando a quantidade de outliers removidos, na tentativa de aumentar o desempenho. Também buscou-se mudar a forma de preenchimento de valores ausentes, mudando da média para inserção de zeros. Após todas as modificações, alcançou-se o desempenho de R2 médio igual a 0.93417 e desvio padrão igual a 0.01102, superando os demais modelos.

Uma das desvantagens dessa abordagem, no entanto, é a demora no tempo de treinamento. A necessidade de treinar múltiplos modelos, muitos deles consideravelmente complexos, e combiná-los aumenta consideravelmente o tempo gasto no treinamento. Isso impacta muito na validação cruzada especialmente, visto que o processo de treinamento é repetido várias vezes.

## Stacking: novas tentativas

Devido ao fato de o prazo de entrega desta atividade ter sido adiado em uma semana, foi adquirido mais tempo para a realização de novas experimentações com técnicas diferentes.

Uma primeira modificação a ser feita foi a aplicação da função logaritmo natural, presente no pacote Numpy, na variável-alvo. Isso é feito de modo a “neutralizar” o leve viés da distribuição de tal variável para valores mais elevados.

Com esse detalhe, os valores se tornam melhor distribuídos em torno de uma média, condição que pode afetar positivamente no desempenho de certos tipos de modelos. Deve-se atentar, por outro lado, na necessidade de aplicar a função exp à saída do modelo para obter o valor predito adequado antes de escrevê-lo no arquivo de saída.

Outra modificação trata da forma como ocorre o preenchimento dos valores ausentes no conjunto de treino. Até então, o preenchimento se dava de acordo com a mesma estratégia para todas as variáveis. No entanto, devido a cada variável possuir um significado distinto, o mais adequado é preencher cada uma de modo a manter o significado desejado que valores ausentes podem possuir.

Assim, tal fase do pré-processamento foi executada coluna a coluna. Alguns valores ausentes, em variáveis numéricas, foram preenchidos com o valor numérico zero, enquanto outros com a moda (no lugar da média aritmética utilizada anteriormente). Já nas variáveis categóricas, o preenchimento se deu com um classe “None” para tais valores.

```
varNone=["PoolQC","MiscFeature","Alley","Fence","FireplaceQu","GarageType","GarageFinish","GarageQual","GarageCond",
         'BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2',"MasVnrType",'MSSubClass']

for i in varNone:
    complete[i] = complete[i].fillna('None')

complete["LotFrontage"] = complete.groupby("Neighborhood")["LotFrontage"].transform(lambda x: x.fillna(x.median()))

var0=['GarageYrBlt', 'GarageArea', 'GarageCars','BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF','TotalBsmtSF', 'BsmtFullBath',
      'BsmtHalfBath',"MasVnrArea"]

for i in var0:
    complete[i] = complete[i].fillna(0)

varMode=['Electrical','KitchenQual','Exterior1st','Exterior2nd','SaleType']

for i in varMode:
    complete[i] = complete[i].fillna(complete[i].mode()[0])
```

**Novo preenchimento de valores ausentes**

Por fim, o grande diferencial implementado na última semana de desenvolvimento foi a técnica de stacking de vários modelos bases. Em resumo, no lugar de uma simples média aritmética das saídas de cada modelo, é gerada uma estrutura que armazena as saídas. Então, é treinado um modelo linear Ridge para determinar a melhor combinação dos modelos.

O modelo usado na otimização da saída é chamado de meta-modelo, à medida que os atributos combinados são chamados de meta-features. Outra diferença considerável é que o conjunto de treinamento é dividido em subconjuntos. Cada subconjunto é utilizado para treinar um modelos-base distinto em uma interação.

Seguindo as definições e implementações disponíveis para esse conceito, foi implementada uma versão que combina modelos-base, além dos modelos já discutidos, KernelRidge, Lasso e linear com regularização Elastic Net conforme sugerido.

```
from sklearn.base import BaseEstimator, RegressorMixin, TransformerMixin, clone
import numpy as np
from sklearn.model_selection import KFold
class StackingAveragedModels(BaseEstimator, RegressorMixin, TransformerMixin):
    def __init__(self,models,meta,n):
        self.models=models
        self.meta=meta
        self.n=n
    def fit(self, X, y):
        self.models2 = [list() for x in self.models]
        self.meta2 = clone(self.meta)
        kfold = KFold(n_splits=self.n,shuffle=True)
        others=np.zeros((X.shape[0], len(self.models)))
        for i, model in enumerate(self.models):
            for trainidx, holdout in kfold.split(X, y):
                self.models2[i].append(clone(model))
                instance.fit(X[trainidx],y[trainidx])
                others[holdout, i]=instance.predict(X[holdout])
        self.meta2.fit(others,y)
        return self
    def predict(self, X):
        meta_features=np.column_stack([
            np.column_stack([model.predict(X) for model in models]).mean(axis=1)
            for models in self.models2])
        return self.meta2.predict(meta_features)

model=StackingAveragedModels(models,RidgeCV(),10)
```

**Implementação de stacking**

## Submetendo ao Kaggle

De modo a enviar os resultados de um modelo à plataforma Kaggle para aferição e inclusão no placar de líderes, foi preparado um script especialmente com tal finalidade chamado “sub.py” (com versões sub2.py.e sub3.py para diferentes famílias de modelos). Eles produzem o arquivo de saída .csv solicitado pelo Kaggle para avaliação. Assim, basta submetê-lo através do site.

A grande diferença aqui é a que não mais temos apenas um conjunto de dados a ser dividido em conjunto de treinamento e conjunto de validação. Passamos a ter o mesmo conjunto de treino, que dessa vez será utilizado em sua totalidade para treinamento, e um conjunto de teste, com observações distintas cujos valores de SalePrice são desconhecidos por nós.

Um obstáculo enfrentado e que demandou grande atenção foi o fato de nem todos os valores variáveis categóricas que apareciam no conjunto de treino apareciam também no conjunto de teste. Por isso, após a binarização delas, os dois conjuntos possuíam um número diferente de atributos, impedindo o uso adequado do modelo.

Para contornar esse problema, os dois conjuntos de dados foram concatenados em um só dataframe utilizando um método do Pandas. Todo o pré-processamento, incluindo a binarização de variáveis categóricas, foi feito em tal conjunto concatenado. Então, concluído o pré-processamento, o conjunto foi novamente dividido no conjunto de treino e de teste com base nos identificadores de cada observação. Também foi necessário remover a coluna SalePrice do conjunto de teste adicionada durante a concatenação.

Além disso, o funcionamento do código consiste em treinar o modelo a ser submetido ao Kaggle utilizando o método .fit com o conjunto de treino e aplicá-lo sobre o conjunto de teste utilizando .predict. É utilizado o método de escrita em CSV.to_csv do dataframe Pandas para gerar o arquivo contendo em cada linha apenas o identificador da observação e o valor predito.

## Resultados

Os modelos que desempenharam médias de R2 mais elevadas foram submetidos à plataforma Kaggle para avaliação seguindo o procedimento explicitado na seção anterior.

A primeira submissão feita foi do modelo linear c normalização Ridge ajustado de maneira automatizada (RidgeCV) com valores ausentes completados com 0. Sua pontuação foi de 0.16834.

A seguir, foi feita tentativa similar, de mesmo pré-processamento, com Gradient-boosted trees, acarretando em pontuação de 0.18015.

Logo depois, foi submetido um modelo idêntico ao primeiro mas com taxa de aprendizagem alfa=15.0. Foi adquirida uma ligeira melhora, alcançando a pontuação de 0.16365. Uma tentativa similar descartando as variáveis LotFrontage, MasVnrArea e GarageYrBlt também foi testada porém com pontuação de 0.16633.

Em seguida, foi submetido o teste com SVM, treinada com os hiperparâmetros mencionados anteriormente. Foi alcançado a melhor pontuação até agora, atingindo 0.13365 e ocupando a posição 1689 do ranking geral da competição.

O modelo BayesianRidge, que teve desempenho excelente na validação cruzada executada localmente no conjunto de treino, acabou tendo desempenho abaixo das expectativas no conjunto de teste. Ele alcançou pontuação de 0.19188 na plataforma. Isso leva a crer que o conjunto de teste pode conter observações muito diferentes das do conjunto de treino, ou ainda, que o conjunto de treino não cobriu toda a variedade de observações que poderia.

O XGBoost puro com 650 estimadores teve pontuação 0.17033 no Kaggle. Já utilizando o ensemble próprio, incluindo o XGBRegressor, foi atingida a pontuação de 0.12942. Após trocar a estratégia de preenchimento de valores ausentes da média para a inserção de zeros, obteve-se pontuação de 0.12627.

Foi feito, ainda um teste com PolynomialFeatures de grau 2. No entanto sua pontuação foi menor: 0.13245. Finalmente, ajustando os outliers removidos no pré-processamento, alcançou-se a melhor pontuação até o momento de 0.11753. Tal valor rendia a posição 630 no ranking do Kaggle.

Já na semana extra após a extensão do prazo, foram feitos os testes com o novo stacking. Este recurso apresentou os melhores resultados do projeto verificados no Kaggle. Foi obtida pontuação de 0.11543, rendendo a posição de número 260 no ranking da plataforma no momento do processamento.

No total, foram realizadas 52 submissões ao Kaggle no decorrer desta atividade. A pontuação de cada submissão é dada pela raiz do erro quadrático médio (RMSE) entre os logaritmos naturais da predição e do valor de referência confidenciais. O perfil utilizado para se inscrever na competição pode ser acessado em: https://www.kaggle.com/lucasbo.

É importante ressaltar que, como o ensemble criado utiliza modelos randômicos, os resultados obtidos podem variar ligeiramente. Estes valores foram os melhores encontrados após algumas submissões. Novos testes não puderam ser efetuados em razão do término do prazo limite para a entrega do trabalho.

## Conclusão

Isso posto, nota-se que os objetivos originais do trabalho foram cumpridos, identificando um modelo com boa capacidade de predição. Não deve-se esperar que exista um modelo capaz de realizar predições com erro nulo para toda e qualquer observação. A qualidade de uma solução está em seu potencial de generalizar o problema é gerar predições aceitavelmente próximas dos valores reais, o que foi alcançado.

Em termos acadêmicos, esta atividade representou uma oportunidade única de colocar em prática os conhecimentos sobre aprendizado de máquina adquiridos no decorrer do curso. Além disso, permitiu o contato com um problema real e de aplicações práticas.

## Referências

[1] Dean De Cock. House Prices: Advanced Regression Techniques. Competição Kaggle. Disponível em: https://www.kaggle.com/c/house-prices-advanced-regression-techniques.

[2] Scikit-learn. Documentação da versão estável 0.19.2. Disponível em: http://scikit-learn.org/stable/modules/classes.html.

[3] Cristoph Gohlke. Repositório de eggs de bibliotecas Python. Universidade da Califórnia, Irvine. Disponível://www.lfd.uci.edu/~gohlke/pythonlibs/.

[4] XGBoost. Documentação oficial. Disponível em: https://xgboost.readthedocs.io/en/latest/python/python_api.html.