PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
NÚCLEO DE EDUCAÇÃO A DISTÂNCIA
Pós-graduação *Lato Sensu* em Ciência de Dados e Big Data

Alencar Lucas Pletsch

Previsão de Preços de Apartamentos em Curitiba-PR

(Bairros Selecionados)

Belo Horizonte
2021

Alencar Lucas Pletsch

Previsão de Preços de Apartamentos em Curitiba-PR

(Bairros Selecionados)

Trabalho de Conclusão de Curso apresentado ao Curso de Especialização em Ciência de Dados e Big Data como requisito parcial à obtenção do título de especialista.

Belo Horizonte

2021

SUMÁRIO

1. Introdução

1.1. Contextualização

É de conhecimento geral que o mercado imobiliário possui grande importância na economia brasileira como um todo e, em especial, na vida íntima das pessoas, afinal a aquisição da “casa própria” é o sonho de muitas delas.

Por isso, foi escolhido um tema diretamente relacionado a esse assunto, qual seja precificação de ativos imobiliários (apartamentos) por meio de análise de regressão linear. Embora seja uma técnica simples, a análise de regressão linear tem grande aplicabilidade prática para a previsão de preços com base em dados de mercado.

No trabalho em questão foram utilizadas duas bases de dados, uma com os dados de apartamentos[1] de alguns bairros da cidade de Curitiba-PR, e, outra com dados econômicos dos bairros, da mesma cidade, extraída do estudo intitulado “*Perfil demográfico e socioeconômico dos bairros agregados de Curitiba*”[2], elaborado por meio de uma parceria entre o Departamento Intersindical de Estatística e Estudos Socioeconômicos (DIEESE) e a Prefeitura de Curitiba.

Espera-se que, após o tratamento e análise adequada, tal conjunto de dados possa servir para prever os preços de apartamentos em Curitiba-PR com certo grau de acerto, de modo a evitar transações por preços que fogem ao padrão normal, reduzindo assim a possibilidade de prejuízos financeiros às partes envolvidas.

1.2. O problema proposto

Nesse trabalho realizaremos uma análise dos dados referentes a preços de imóveis (apartamentos) na cidade de Curitiba-PR para fins de precificação. Na sequência, por meio da técnica dos 5-Ws, detalharemos o problema para melhor compreensão e delimitação do tema.

---

[1] Disponível em leg.ufpr.br/~walmes/data/ap_venda7bairros_cwb_210314.txt

[2] Disponível em bairros.pdf (coreconpr.gov.br)

a) Por que este problema é importante (why)?

A importância desse problema repousa no impacto em termos de segurança e satisfação que a aquisição de um imóvel próprio proporciona as pessoas em geral. Os imóveis sempre estiveram no imaginário das pessoas como “um sonho a ser conquistado”, muito disso em função do valor dispendido para a sua aquisição. Como a renda média das pessoas é baixa, muitas delas levam a vida inteira para adquirir um imóvel, caso obtenham êxito nessa empreitada.

Enquanto buscam esse objetivo, convivem com riscos relacionados ao não pagamento de aluguel, como despejo e inscrição em cadastros de inadimplentes, riscos relacionados à redução da renda em função da necessidade de pagamento de aluguel, riscos relacionados à perda de liberdade em sentido amplo nos casos de moradia conjunta com familiares etc. Esses riscos podem afetar a autoestima das pessoas causando-lhes danos à saúde como ansiedade e depressão, e, também, desestabilizar a harmonia familiar, motivando em muitos casos agressões e divórcios com danos emocionais para filhos.

Na ânsia de se livrar desses riscos muitas pessoas entram no mercado imobiliário demandando imóveis sem ter, entretanto, a expertise necessária para fazer a correta avaliação do preço ofertado. Nesses casos, em função do desconhecimento natural e da urgência na aquisição do imóvel, são facilmente influenciáveis com os discursos já conhecidos de que “*nunca se perde dinheiro com imóveis”* e de que *“adquirir imóveis é um investimento”*, acarretando compras desvantajosas e prejuízos financeiros que podem expor as pessoas aos mesmos riscos que pretendiam evitar.

Do lado dos ofertantes de imóveis, precificar adequadamente evitaria influências de corretores e imobiliárias tendentes a fechar a negociação no menor prazo possível a fim de receber o quanto antes a comissão que lhe é devida. Também evitaria o risco de incorrer no conceito de ilusão monetária e achar que está ganhando dinheiro quando, realmente não está, pois muitas pessoas não dispõem do conhecimento necessário para efetuar cálculos de atualização do capital, considerando taxas de inflação e taxas de juros de referência.

b) De quem são os dados analisados e como foram coletados (who)?

Os dados foram obtidos de duas fontes distintas conforme abaixo:

- Apartamentos à venda por bairro em Curitiba: Este conjunto de dados faz parte da disciplina "Manipulação e Visualização de Dados", do Laboratório de Estatística e Geoinformação da Universidade Federal do Paraná (LEG/UFPR) e foi apresentado pelo Prof. Walmes M. Zeviani, em 10/06/2019, aos seus alunos como conjuntos de dados para praticar com a linguagem R[3]. Segundo consta do enunciado da questão "*São dados de apartamentos à venda em Curitiba anunciados em site de imóveis. As informações são de imóveis em 7 bairros de Curitiba e descrevem cada imóvel em termos de preço de venda, metragem e número de cômodos*."

- Área em Km2, população e habitantes por Km2, segundo bairro agregado: Este conjunto de dados consta do Anexo I do estudo intitulado "*Perfil demográfico e socioeconômico dos bairros agregados de Curitiba*"[4], elaborado por meio de uma parceria entre o Departamento Intersindical de Estatística e Estudos Socioeconômicos (DIEESE) e a Prefeitura de Curitiba.

c) Qual é o objeto desta análise (what)?

Por meio desse trabalho analisaremos os dados de preços de imóveis (apartamentos) de sete bairros da cidade de Curitiba-PR, bem como dados referentes ao perfil demográfico (densidade populacional) para fins de precificação.

Para tanto, utilizaremos os algoritmos de regressão linear simples disponíveis nas bibliotecas Scikit-learn[5] e Statsmodel[6].

d) Em que ambiente geográfico esta análise está inserida (where)?

O escopo dessa análise está restrito aos conjuntos de dados que se referem à parte dos bairros de Curitiba-PR, ou seja, não estão disponíveis os dados de todos os bairros da cidade.

---

[3] Disponível em Conjuntos de dados para praticar (ufpr.br)
[4] Disponível em bairros.pdf (coreconpr.gov.br)
[5] Fonte: 1. Supervised learning — scikit-learn 0.24.1 documentation (scikit-learn.org)
[6] Fonte: Introduction — statsmodels

e) Qual o período compreendido nesta análise (when)?

Com relação ao período analisado, verificou-se que os dados obtidos no estudo do DIEESE em parceria com a Prefeitura de Curitiba, embora realizado em fevereiro de 2016, referem-se ao Censo/IBGE de 2010.

Já os dados de preço de apartamento obtido do LEG/UFPR, embora a atividade possua como data o ano de 2019, não há informação sobre o período da coleta dos dados. Como o arquivo de texto com os dados possui o nome *"ap_venda7bairros_cwb_210314.txt"*, deduz-se que seja de março de 2014.

Apesar de haver um hiato temporal entre os dados do censo de 2010 e a coleta de preços de 2014, acreditamos que tenha se mantido a proporção da densidade populacional entre os bairros distintos, não afetando o resultado da análise.

2. Coleta de Dados

Os dados utilizados na presente análise são oriundos de duas bases diferentes, uma com os preços de imóveis (apartamentos) de sete bairros de Curitiba-PR e outra com dados demográficos da mesma região. A seguir apresentaremos maiores detalhes de cada base de dados.

2.1. Apartamentos à venda por bairro em Curitiba (Apartamentos):

Como já foi mencionado anteriormente, este conjunto de dados faz parte da disciplina "Manipulação e Visualização de Dados", do Laboratório de Estatística e Geoinformação da Universidade Federal do Paraná (LEG/UFPR) e foi apresentado pelo Prof. Walmes M. Zeviani, em 10/06/2019, aos seus alunos como conjuntos de dados para praticar com a linguagem R[7]. Segundo consta do enunciado da questão "*São dados de apartamentos à venda em Curitiba anunciados em site de imóveis. As informações são de imóveis em 7 bairros de Curitiba e descrevem cada imóvel em termos de preço de venda, metragem e número de cômodos*."

Estes dados foram obtidos no site da disciplina supracitada no seguinte endereço eletrônico: *leg.ufpr.br/~walmes/data/ap_venda7bairros_cwb_210314.txt*, em 05/01/2021. Como pode ser percebido, trata-se de um arquivo de texto (.txt), sendo importado para o Jupyter Notebook por meio da biblioteca Pandas[8] usando os seguintes comandos:

Figura 1 Código para leitura do dataset

```
In [99]: #Dados = pd.read_csv('http://leg.ufpr.br/~walmes/data/ap_venda7bairros_cwb_210314.txt', sep='\t')
         Apart = pd.read_csv('ap_venda7bairros_cwb_210314.txt', sep='\t')
         Apart.shape

Out[99]: (4470, 7)
```

No caso foi optado pelo download do arquivo e posterior leitura pelo Pandas, pois havia o receio de que a página não estivesse disponível em algum momento futuro. Porém, poderia ter sido feita a leitura do dataset diretamente na página em que está hospedado por meio do comando que está comentado (#).

[7] Disponível em Conjuntos de dados para praticar (ufpr.br)
[8] Disponível em pandas - Python Data Analysis Library (pydata.org)

O dataset possui o formato de 4.470 linhas por 7 colunas e está estruturado da seguinte forma:

Tabela 1 Estrutura do dataset Apartamentos

Nome da coluna/campo	**Descrição**	**Tipo**
preco	Valor anunciado para venda do apartamento	int64
area	Área do apartamento em metros quadrados	float64
quartos	Quantidade de quartos incluindo suítes	float64
banheiros	Quantidade de banheiros incluindo banheiros das suítes	float64
vagas	Quantidade de vagas para estacionamento de veículos	float64
suites	Quantidade de suítes (quartos com banheiros)	float64
bairro	Nome do bairro em que está localizado o apartamento	object

Abaixo segue o *head* do *dataframe* do Pandas:

Figura 2 - Head do dataframe Apartamentos

	preco	area	quartos	banheiros	vagas	suites	bairro
0	286000	63.00	2.0	2.0	1.0	1.0	portao
1	328000	75.00	3.0	2.0	1.0	1.0	portao
2	370000	62.77	2.0	2.0	2.0	1.0	portao
3	295000	62.88	2.0	2.0	1.0	1.0	portao
4	260000	75.00	3.0	2.0	1.0	1.0	portao

O campo que será utilizado para o *join* dos *dataframes* é o denominado “bairro”. Assim, a relação entre os *datasets* se dá pelo atributo bairro.

2.2. Área em Km², população e habitantes por Km², segundo bairro agregado (Bairros)

Retomando o já mencionado anteriormente, este conjunto de dados consta do Anexo I do estudo intitulado "*Perfil demográfico e socioeconômico dos bairros agregados de Curitiba*"[9], elaborado por meio de uma parceria entre o Departamento Intersindical de Estatística e Estudos Socioeconômicos (DIEESE) e a Prefeitura de Curitiba. Embora o estudo seja de fevereiro de 2016, referem-se ao Censo/IBGE de 2010.

Tal estudo foi obtido na internet em 07/01/2021, no endereço eletrônico *bairros.pdf (coreconpr.gov.br)*, página do Conselho Regional de Economia do Paraná. Diferentemente do *dataset* anterior, este é uma tabela intitulada Anexo I dentro de um arquivo PDF (vide figura abaixo).

Figura 3 - Anexo I do estudo do DIEESE

DIEESE

ANEXO I
Área em Km², população e habitantes por Km², segundo bairro agregado. Curitiba, 2010

Bairro agregado	População		Área (KM2)		Hab. / Km2
	Nº absoluto	Part. %	Nº absoluto	Part. %	
Centro	37.234	2,1	3.310	0,8	11,2
Água Verde	51.461	2,9	4.850	1,1	10,6
Sítio Cercado	115.584	6,6	11.252	2,6	10,3
Batel e Bigorrilho	39.234	2,2	4.633	1,1	8,5
Cajuru	96.170	5,5	11.781	2,7	8,2
Fazendinha	28.101	1,6	3.779	0,9	7,4
Novo Mundo	44.056	2,5	6.035	1,4	7,3
Portão	42.038	2,4	5.838	1,3	7,2
Vila Izabel e Santa Quitéria	24.045	1,4	3.364	0,8	7,1
Capão Razo	36.067	2,1	5.062	1,2	7,1
Centro Cívico e Juveve	21.909	1,3	3.080	0,7	7,1

Para a leitura dos dados foi utilizada a biblioteca Tabula[10], a qual permite ler os dados em arquivos PDF e convertê-los em *dataframes* do Pandas. Abaixo segue

[9] Disponível em bairros.pdf (coreconpr.gov.br)
[10] Disponível em tabula-py · PyPI

fragmento do código utilizado para leitura e manipulação do *dataframe*, manipulação esta necessária pois na leitura inicial o dataframe ficou desfigurado.

Figura 4 - Leitura e manipulação do dataset Bairros

In [103]:
```
#!pip install tabula-py
from tabula import read_pdf
Bairros = read_pdf("Bairros_DIEESE.pdf", pages='64', stream=True, area='left')
Bairros = Bairros[0]
Bairros = Bairros.drop(['Unnamed: 1'],axis=1)
Bairros = Bairros.iloc[5:46].reset_index()
Bairros = Bairros['Unnamed: 0'].str.rsplit(n=5)

Bairros = pd.DataFrame(Bairros)

Bairros['Bairro'] = Bairros['Unnamed: 0'][5][0]
Bairros['População'] = Bairros['Unnamed: 0'][5][1]
Bairros['População(%)']= Bairros['Unnamed: 0'][5][2]
Bairros['Área(km2)']= Bairros['Unnamed: 0'][5][3]
Bairros['Área(%)']= Bairros['Unnamed: 0'][5][4]
Bairros['Habitantes/km2']= Bairros['Unnamed: 0'][5][5]


for i in range(0,len(Bairros)):
    Bairros['Bairro'][i] = Bairros['Unnamed: 0'][i][0]
    Bairros['População'][i] = Bairros['Unnamed: 0'][i][1]
    Bairros['População(%)'][i] = Bairros['Unnamed: 0'][i][2]
    Bairros['Área(km2)'][i] = Bairros['Unnamed: 0'][i][3]
    Bairros['Área(%)'][i] = Bairros['Unnamed: 0'][i][4]
    Bairros['Habitantes/km2'][i] = Bairros['Unnamed: 0'][i][5]

Bairros = Bairros.drop(['Unnamed: 0'], axis=1)
Bairros = Bairros.drop([40])
Bairros.head()
```

Após o tratamento acima demonstrado o *dataframe* ficou com o formato de 40 linhas e 6 colunas, sendo todos as colunas tipo *object.* Desse modo, foram necessários mais alguns ajustes no dataframe como correção de nomes[11] e transformações dos tipos de atributos conforme código abaixo:

[11] A função “corrigir_nomes” foi obtida no seguinte site python - Como retirar caractere especial e ponto de coluna string de um data frame? - Stack Overflow em Português

Figura 5 - Transformação do dataframe Bairros

```
In [260]:
def corrigir_nomes(nome):
    nome = nome.replace('.', '').replace(',', '.')
    return nome
Bairros['População'] = Bairros['População'].astype('string').apply(corrigir_nomes)
Bairros['População(%)'] = Bairros['População(%)'].astype('string').apply(corrigir_nomes)
Bairros['Área(km2)'] = Bairros['Área(km2)'].astype('string').apply(corrigir_nomes)
Bairros['Área(%)'] = Bairros['Área(%)'].astype('string').apply(corrigir_nomes)
Bairros['Habitantes/km2'] = Bairros['Habitantes/km2'].astype('string').apply(corrigir_nomes)
Bairros.head()
```

Out[260]:

	Bairro	População	População(%)	Área(km2)	Área(%)	Habitantes/km2
0	Centro	37234	2.1	3310	0.8	11.2
1	Água Verde	51461000000000006	2.9	4850	1.1	10.6
2	Sítio Cercado	115584	6.6	11252	2.6	10.3
3	Batel e Bigorrilho	39234	2.2	4633	1.1	8.5
4	Cajuru	9617	5.5	11781	2.7	8.2

```
In [261]:
Bairros['População'] = pd.to_numeric(Bairros['População'])
Bairros['População(%)'] = pd.to_numeric(Bairros['População(%)'])
Bairros['Área(km2)'] = pd.to_numeric(Bairros['Área(km2)'])
Bairros['Área(%)'] = pd.to_numeric(Bairros['Área(%)'])
Bairros['Habitantes/km2'] = pd.to_numeric(Bairros['Habitantes/km2'])

Bairros.dtypes
```

```
Out[261]:
Bairro             object
População           int64
População(%)      float64
Área(km2)           int64
Área(%)           float64
Habitantes/km2    float64
dtype: object
```

Na figura acima podemos observar o *head* do *dataframe* Bairros. Na tabela abaixo vemos a estrutura dos campos do *dataframe* Bairros:

Tabela 2 Estrutura do dataframe Bairros

Nome da coluna/campo	**Descrição**	**Tipo**
Bairro	Nome do bairro agregado em que está localizado o apartamento	object
População	População do bairro agregado	int64
População(%)	População percentual do bairro agregado	float64
Área(km2)	Área do bairro agregado	int64
Área(%)	Área percentual do bairro agregado	float64
Habitantes/km2	Densidade populacional do bairro agregado	float64

Como pode ser percebido acima, o *dataset* Bairros utiliza o conceito de “Bairro Agregado” e refere-se à junção de alguns bairros circunvizinhos de Curitiba-PR para a realização do estudo.

Entendemos que essa variação do conceito não prejudicará o resultado da análise, sendo, então, considerado como se fosse o mesmo bairro do *dataset* Apartamento.

O relacionamento dos dois *datasets* será por meio dos atributos ‘bairro’ (dataframe = Apart) e ‘Bairro’ (dataframe = Bairros), os quais basicamente tem o nome dos bairros de Curitiba-PR. No próximo tópico discorreremos sobre os comandos utilizados para a realização da junção dos *dataframes.*

3. Processamento/Tratamento de Dados

Embora já tenhamos no tópico anterior trabalhado parcialmente os conjuntos de dados para fins de leitura e estruturação dos *dataframes*, agora vamos tratar da parte de processamento e tratamento dos dados a partir da junção dos *dataframes*.

3.1. *Join* dos *dataframes*

Primeiro, relembramos que faremos o *join* pelas colunas que contém o nome dos bairros em ambos os *datasets*. Verificamos que há diferença na descrição dos bairros em cada *dataset*, o que inviabilizaria a correspondência adequada dos registros.

Figura 6 - Descrição dos nomes dos bairros em cada dataset

```
In [12]: Bairros['Bairro'].unique()

Out[12]: array(['Centro', 'Água Verde', 'Sítio Cercado', 'Batel e Bigorrilho',
                'Cajuru', 'Fazendinha', 'Novo Mundo', 'Portão',
                'Vila Izabel e Santa Quitéria', 'Capão Razo',
                'Centro Cívico e Juveve', 'Bairro Alto', 'Cabral', 'Xaxim',
                'Boa vista', 'Guaíra e Fanny', 'Uberaba', 'Pinheirinho',
                'Tatuquara', 'Boqueirão e Hauer', 'Alto Boqueirão',
                'Atuba e Tinguí', 'Alto da XV expandido', 'Rebouças expandido',
                'Guabirotuba e Jardim das Américas', 'Barreirinha e Cachoeira',
                'Cidade Industrial', 'Capão da Embuia e Tarumã', 'Pilarzinho',
                'São Braz e Santo Inácio', 'Campo Comprido',
                'São Francisco expandido', 'Bacacheri',
                'Campina do Siqueira expandido', 'Santa Cândida',
                'Abranches expandido', 'Santa Felicidade expandido',
                'Campo do Santana e Caximba', 'Ganchinho e Umbará', 'CIC norte'],
               dtype=object)

In [13]: Apart['bairro'].unique()

Out[13]: array(['portao', 'agua-verde', 'centro', 'ecoville', 'batel', 'cabral',
                'cristo-rei'], dtype=object)
```

Dessa forma, foi necessário criar a coluna “bairro” no dataset “Bairro” e ajustar a descrição dos bairros para haver a correspondência exata com a coluna “bairro” do dataset “Apart”, o que foi realizado conforme o código a seguir:

Figura 7 - Criação da coluna "bairro" e ajustes dos registros com o nome dos bairros

In [14]:

```
Bairros['bairro']=Bairros['Bairro'].str.lower()
Bairros['bairro']=Bairros['bairro'].replace('água verde','agua-verde'
                                            ).replace('batel e bigorrilho','batel'
                                                      ).replace('alto da xv expandido','cristo-rei'
                                                                ).replace('campina do siqueira expandido','ecoville'
                                                                          ).replace('portão','portao')
Bairros.head()
```

Out[14]:

	Bairro	População	População(%)	Área(km2)	Área(%)	Habitantes/km2	bairro
0	Centro	37234	2.1	3310	0.8	11.2	centro
1	Água Verde	51461	2.9	4850	1.1	10.6	agua-verde
2	Sítio Cercado	115584	6.6	11252	2.6	10.3	sítio cercado
3	Batel e Bigorrilho	39234	2.2	4633	1.1	8.5	batel
4	Cajuru	96170	5.5	11781	2.7	8.2	cajuru

Na sequência foi realizado o join dos dataframes “Apart” e “Bairros” em um novo dataframe denominado “Dados”. Sendo, posteriormente, alterado o nome das colunas “Habitantes/km2” e “Área(km2)” para “Densidade_populacional” e “Area_bairro”, respectivamente. Tal alteração dos nomes foi necessária para evitar erros nos algoritmos de regressão executados posteriormente.

Também, foi eliminada a coluna “Bairro” pois esta não seria mais de utilidade para a análise. Seguem os códigos:

Figura 8 - Join dos Dataframes

In [15]:

```
#Join dos datasets para análise
Dados = Apart.merge(Bairros, left_on='bairro', right_on='bairro')
Dados.rename(columns={'Habitantes/km2': 'Densidade_populacional'}, inplace = True)
Dados.rename(columns={'Área(km2)': 'Area_bairro'}, inplace = True)
Dados = Dados.drop(['Bairro'], axis=1)
Dados.head()
```

Out[15]:

	preco	area	quartos	banheiros	vagas	suites	bairro	População	População(%)	Area_bairro	Área(%)	Densidade_populacional
0	286000	63.00	2.0	2.0	1.0	1.0	portao	42038	2.4	5838	1.3	7.2
1	328000	75.00	3.0	2.0	1.0	1.0	portao	42038	2.4	5838	1.3	7.2
2	370000	62.77	2.0	2.0	2.0	1.0	portao	42038	2.4	5838	1.3	7.2
3	295000	62.88	2.0	2.0	1.0	1.0	portao	42038	2.4	5838	1.3	7.2
4	260000	75.00	3.0	2.0	1.0	1.0	portao	42038	2.4	5838	1.3	7.2

Depois foi acrescentada a coluna “preco_metro” com o valor do cálculo do preço/m² e convertido o tipo de dados da coluna “bairro” para “category” em função de erros na execução do algoritmo de regressão linear.

Figura 9 - Inclusão da coluna "preco_metro" e mudança do tipo de dados da coluna "bairro"

```
In [16]:
#Acrescentando a coluna de preço/metro²
Dados['preco_metro'] = Dados['preco']/Dados['area']

In [17]:
#convertendo bairro para categorical
Dados['bairro'] = Dados['bairro'].astype('category')
Dados.dtypes

Out[17]: preco                      int64
         area                     float64
         quartos                  float64
         banheiros                float64
         vagas                    float64
         suites                   float64
         bairro                  category
         População                  int64
         População(%)             float64
         Area_bairro                int64
         Área(%)                  float64
         Densidade_populacional   float64
         preco_metro              float64
         dtype: object
```

3.2. *Tratamento de Missing Values*

Com os datasets consolidados em um só vamos analisar os valores faltantes (*missing values*) e fazer o tratamento adequado.

Verificamos no dataset “Dados” uma grande quantidade de missing values, os quais são oriundos do dataset “Apart” que continha dados de anúncios. Provavelmente, a ausência desses valores seja decorrente de anúncios preenchidos de forma equivocada e que prejudica em demasia a análise dos dados.

Abaixo podemos verificar a quantidade de missing values por campos dos registros:

Figura 10 - Missing values

```
In [19]:
Dados.isna().sum()

Out[19]: preco                      0
         area                       0
         quartos                   48
         banheiros               1030
         vagas                    533
         suites                  1161
         bairro                     0
         População                  0
         População(%)               0
         Area_bairro                0
         Área(%)                    0
         Densidade_populacional     0
         preco_metro                0
         dtype: int64
```

Analisando os registros com missing values tanto na coluna “suites” quanto na coluna “banheiros”, as quais são as colunas com maior quantidade de dados ausentes, contabilizamos 486 registros.

Figura 11 - Dados missing nas colunas "banheiros" e "suites", simultaneamente

In [18]:
```
Dados.loc[(Dados['suites'].isna()==True) & (Dados['banheiros'].isna()==True)]
```

Out[18]:

	preco	area	quartos	banheiros	vagas	suites	bairro	População	População(%)	Area_bairro	Área(%)	Densidade_populacional	preco_metro
55	250000	66.95	3.0	NaN	1.0	NaN	portao	42038	2.4	5838	1.3	7.2	3734.129948
84	689691	104.00	3.0	NaN	2.0	NaN	portao	42038	2.4	5838	1.3	7.2	6631.644231
93	220000	58.00	3.0	NaN	NaN	NaN	portao	42038	2.4	5838	1.3	7.2	3793.103448
97	295000	109.00	3.0	NaN	1.0	NaN	portao	42038	2.4	5838	1.3	7.2	2706.422018
106	333195	75.00	3.0	NaN	1.0	NaN	portao	42038	2.4	5838	1.3	7.2	4442.600000
...	...	...	...	...	...	...	...	...	...	...	...	...	...
4414	222921	58.00	1.0	NaN	2.0	NaN	cristo-rei	37565	2.1	8896	2.0	4.2	3843.465517
4433	221433	62.00	1.0	NaN	2.0	NaN	cristo-rei	37565	2.1	8896	2.0	4.2	3571.500000
4440	203391	55.00	1.0	NaN	1.0	NaN	cristo-rei	37565	2.1	8896	2.0	4.2	3698.018182
4441	219945	61.00	1.0	NaN	2.0	NaN	cristo-rei	37565	2.1	8896	2.0	4.2	3605.655738
4447	706000	121.00	NaN	NaN	2.0	NaN	cristo-rei	37565	2.1	8896	2.0	4.2	5834.710744

486 rows × 13 columns

Apesar de haver técnicas que sugerem a substituição dos missing values por valores como média, mediana etc., entendemos que, ao fazer isso, estaríamos distorcendo de tal maneira os registros que poderiam ocorrer aberrações como apartamentos com 1 quarto e três banheiros.

Assim, dado que, mesmo excluindo os registros com missing values, ainda teríamos uma quantidade relevante de registros para análise (2.652 registros) preferimos manter a integridade destes e evitar distorções que possam prejudicar o resultado da análise. Abaixo segue o código utilizado para exclusão dos missing values.

Figura 12 - Exclusão de missing values

In [20]:
```
Dados = Dados.dropna()
Dados.shape
```

Out[20]:
```
(2652, 13)
```

In [21]:
```
Dados.isna().sum()
```

Out[21]:
```
preco                   0
area                    0
quartos                 0
banheiros               0
vagas                   0
suites                  0
bairro                  0
População               0
População(%)            0
Area_bairro             0
Área(%)                 0
Densidade_populacional  0
preco_metro             0
dtype: int64
```

3.3. *Dados Duplicados*

Após análise dos dados duplicados, verificamos a existência de 419 registros nessa situação.

Figura 13 - Dados duplicados

3.2 Dados Duplicados

```
In [104]: Dados[Dados.duplicated(keep=False)]
```

Out[104]:

	preco	area	quartos	banheiros	vagas	suites	bairro	População	População(%)	Area_bairro	Área(%)	Densidade_populacional	preco_metro	preco_
0	286000	63.0	1.0	1.0	1.0	1.0	6	42038	2.4	5838	1.3	7.2	4539.682540	286000
1	328000	75.0	2.0	1.0	1.0	1.0	6	42038	2.4	5838	1.3	7.2	4373.333333	328000
5	328000	75.0	2.0	1.0	1.0	1.0	6	42038	2.4	5838	1.3	7.2	4373.333333	328000
6	328000	75.0	2.0	1.0	1.0	1.0	6	42038	2.4	5838	1.3	7.2	4373.333333	328000
7	790000	129.0	0.0	2.0	2.0	3.0	6	42038	2.4	5838	1.3	7.2	6124.031008	790000
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
4366	420000	83.0	2.0	1.0	1.0	1.0	4	37565	2.1	8896	2.0	4.2	5060.240964	420000
4371	831631	143.0	3.0	2.0	2.0	1.0	4	37565	2.1	8896	2.0	4.2	5815.601399	831631
4374	684900	123.0	2.0	2.0	2.0	1.0	4	37565	2.1	8896	2.0	4.2	5568.292683	684900
4383	300000	68.0	2.0	1.0	1.0	1.0	4	37565	2.1	8896	2.0	4.2	4411.764706	300000
4391	831631	143.0	3.0	2.0	2.0	1.0	4	37565	2.1	8896	2.0	4.2	5815.601399	831631

419 rows × 16 columns

Embora em alguns casos seja possível verificar que o registro duplicado seja referente ao mesmo imóvel, essa não é a regra presente no dataset. Como são anúncios é comum que o vendedor renove a publicação com certa frequência, mas, também, pode ocorrer que alguns apartamentos diferentes tenham as mesmas características, inclusive o preço. Isso ocorre principalmente nos anúncios de menor valor, os quais se comportam como se fossem “comodities” em função da ausência de diferenciação.

Assim, em função da dificuldade de distinção de qual registro seria decorrente de duplicatas ou decorrente de anúncios idênticos, e, para não perder mais registros, optou-se por manter as duplicatas no dataset para fins de análise.

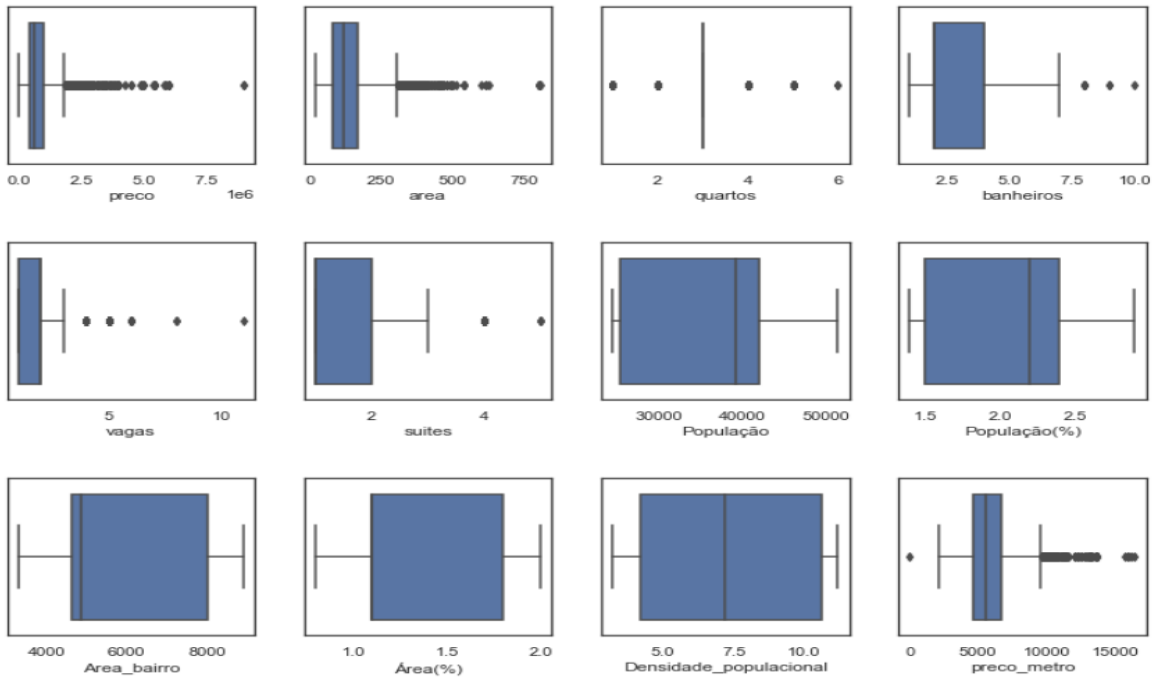
3.4. *Tratamento de outliers*

Agora vamos partir para a análise de outliers. Nos boxplots abaixo, gerados por meio das bibliotecas matplotlib.pyplot[12] e seaborn[13], podemos verificar a existência de

[12] Documentação disponível em matplotlib.pyplot — Matplotlib 3.4.1 documentation
[13] Documentação disponível em seaborn: statistical data visualization — seaborn 0.11.1 documentation (pydata.org)

outliers nas variáveis “preco”, “area”, “quartos”, “banheiros”, “vagas”, “suítes” e “preco_metro”.

Figura 14 - Boxplot atributos do dataframe Dados



O código para gerar os boxplot pode ser visualizado abaixo.

Figura 15 - Código para gerar boxplot

```
In [150]: plt.figure(figsize = (15, 10))
          plt.subplot(341)
          sns.boxplot(x='preco',data = Dados, orient='h')
          plt.subplot(342)
          sns.boxplot(x='area',data = Dados, orient='h')
          plt.subplot(343)
          sns.boxplot(x='quartos',data = Dados, orient='h')
          plt.subplot(344)
          sns.boxplot(x='banheiros',data = Dados, orient='h')
          plt.subplot(345)
          sns.boxplot(x='vagas',data = Dados, orient='h')
          plt.subplot(346)
          sns.boxplot(x='suites',data = Dados, orient='h')
          plt.subplot(347)
          sns.boxplot(x='População',data = Dados, orient='h')
          plt.subplot(348)
          sns.boxplot(x='População(%)',data = Dados, orient='h')
          plt.subplot(349)
          sns.boxplot(x='Area_bairro',data = Dados, orient='h')
          plt.subplot(3,4,10)
          sns.boxplot(x='Área(%)',data = Dados, orient='h')
          plt.subplot(3,4,11)
          sns.boxplot(x='Densidade_populacional',data = Dados, orient='h')
          plt.subplot(3,4,12)
          sns.boxplot(x='preco_metro',data = Dados, orient='h')

          plt.subplots_adjust(bottom=0.1, right=0.8, top=0.9, hspace=0.5)
          plt.show()
```

Como já exposto anteriormente, optou-se por modificar o mínimo possível o conjunto de dados para evitar distorções. Assim, vamos verificar inicialmente as variáveis quartos, banheiros, vagas e suítes.

a) Quartos:

Por meio do Pandas podemos agrupar os dados pela variável “quartos” e obter a contagem dos valores:

Figura 16 - Contagem de apartamentos pelo número de quartos

```
In [24]: # variáveis quartos
         Dados.groupby(['quartos']).count()
```

Out[24]:

quartos	preco	area	banheiros	vagas	suites	bairro	População	População(%)	Area_bairro	Área(%)	Densidade_populacional	preco_metro
1.0	54	54	54	54	54	54	54	54	54	54	54	54
2.0	517	517	517	517	517	517	517	517	517	517	517	517
3.0	1499	1499	1499	1499	1499	1499	1499	1499	1499	1499	1499	1499
4.0	558	558	558	558	558	558	558	558	558	558	558	558
5.0	22	22	22	22	22	22	22	22	22	22	22	22
6.0	2	2	2	2	2	2	2	2	2	2	2	2

Pelo boxplot, apenas os apartamentos com três quartos não são classificados como outliers. Isso se deve à predominância de apartamentos com essa quantidade de quartos. Entretanto, como a base já está reduzida, não podemos desprezar os apartamentos com menos de seis quartos.

Já com 6 quartos temos dois registros, vamos analisá-los:

Figura 17 - registros com 6 quartos

```
In [25]: #registros com 6 quartos
         Dados.loc[Dados['quartos'] >= 6]
```

Out[25]:

	preco	area	quartos	banheiros	vagas	suites	bairro	População	População(%)	Area_bairro	Área(%)	Densidade_populacional	preco_metro
1009	1080000	243.0	6.0	4.0	2.0	1.0	agua-verde	51461	2.9	4850	1.1	10.6	4444.444444
3056	1500000	390.0	6.0	6.0	2.0	2.0	batel	39234	2.2	4633	1.1	8.5	3846.153846

Como são apenas dois registros vamos removê-los da base para reduzir as distorções.

Figura 18 - Remoção dos registros com mais de 6 quartos

```
In [26]: #removendo os registros com mais de 5 quartos
         Dados = Dados.loc[Dados['quartos'] < 6]

In [27]: # variáveis quartos
         Dados.groupby(['quartos']).count()
```

Out[27]:

quartos	preco	area	banheiros	vagas	suites	bairro	População	População(%)	Area_bairro	Área(%)	Densidade_populacional	preco_metro
1.0	54	54	54	54	54	54	54	54	54	54	54	54
2.0	517	517	517	517	517	517	517	517	517	517	517	517
3.0	1499	1499	1499	1499	1499	1499	1499	1499	1499	1499	1499	1499
4.0	558	558	558	558	558	558	558	558	558	558	558	558
5.0	22	22	22	22	22	22	22	22	22	22	22	22

b) Banheiros:

Novamente, por meio do Pandas vamos agrupar a variável “banheiros” e obter a contagem dos valores:

Figura 19 - Contagem de apartamentos pelo número de banheiros

```
In [28]: # variáveis banheiros
         Dados.groupby(['banheiros']).count()
```

Out[28]:

banheiros	preco	area	quartos	vagas	suites	bairro	População	População(%)	Area_bairro	Área(%)	Densidade_populacional	preco_metro
1.0	409	409	409	409	409	409	409	409	409	409	409	409
2.0	972	972	972	972	972	972	972	972	972	972	972	972
3.0	548	548	548	548	548	548	548	548	548	548	548	548
4.0	333	333	333	333	333	333	333	333	333	333	333	333
5.0	275	275	275	275	275	275	275	275	275	275	275	275
6.0	78	78	78	78	78	78	78	78	78	78	78	78
7.0	27	27	27	27	27	27	27	27	27	27	27	27
8.0	5	5	5	5	5	5	5	5	5	5	5	5
9.0	2	2	2	2	2	2	2	2	2	2	2	2
10.0	1	1	1	1	1	1	1	1	1	1	1	1

Pelo boxplot, apenas os apartamentos com menos de sete banheiros não são classificados como outliers. Mais uma vez, como a base já está reduzida, não podemos desprezar os apartamentos com menos de oito banheiros.

Já com oito ou mais banheiros temos oito registros, vamos analisá-los:

Figura 20 - registros com 8 ou mais banheiros

```
In [61]: #registros com 8 ou mais banheiros
         Dados.loc[Dados['banheiros'] >= 8]
```

Out[61]:

	preco	area	quartos	banheiros	vagas	suites	bairro	População	População(%)	Area_bairro	Área(%)	Densidade_populacional	preco_metro
2349	9000000	800.0	4.0	8.0	8.0	4.0	ecoville	25493	1.5	8007	1.8	3.2	11250.000000
2482	6000000	802.0	4.0	9.0	6.0	4.0	ecoville	25493	1.5	8007	1.8	3.2	7481.296758
2915	2650000	449.0	5.0	8.0	5.0	5.0	batel	39234	2.2	4633	1.1	8.5	5902.004454
3304	830000	227.0	4.0	9.0	2.0	4.0	batel	39234	2.2	4633	1.1	8.5	3656.387665
3332	1950000	330.0	4.0	10.0	4.0	4.0	batel	39234	2.2	4633	1.1	8.5	5909.090909
3665	6017000	452.0	4.0	8.0	5.0	4.0	cabral	24556	1.4	3930	0.9	6.2	13311.946903
3667	6017000	452.0	4.0	8.0	5.0	4.0	cabral	24556	1.4	3930	0.9	6.2	13311.946903
4020	2200000	400.0	4.0	8.0	5.0	4.0	cabral	24556	1.4	3930	0.9	6.2	5500.000000

Como são apenas oito registros vamos removê-los da base para reduzir as distorções.

Figura 21 - Remoção dos registros com 8 ou mais banheiros

```
In [62]: #removendo os registros com mais de 8 banheiros
         Dados = Dados.loc[Dados['banheiros'] < 8]

In [63]: # variáveis banheiros
         Dados.groupby(['banheiros']).count()
```

Out[63]:

banheiros	preco	area	quartos	vagas	suites	bairro	População	População(%)	Area_bairro	Área(%)	Densidade_populacional	preco_metro
1.0	409	409	409	409	409	409	409	409	409	409	409	409
2.0	972	972	972	972	972	972	972	972	972	972	972	972
3.0	548	548	548	548	548	548	548	548	548	548	548	548
4.0	333	333	333	333	333	333	333	333	333	333	333	333
5.0	275	275	275	275	275	275	275	275	275	275	275	275
6.0	78	78	78	78	78	78	78	78	78	78	78	78
7.0	27	27	27	27	27	27	27	27	27	27	27	27

c) Vagas:

Agora vamos agrupar a variável “vagas” e obter a contagem dos valores:

Figura 22 - Contagem de apartamentos pelo número de vagas

```
In [64]: # variáveis vagas
         Dados.groupby(['vagas']).count()
```

Out[64]:

vagas	preco	area	quartos	banheiros	suites	bairro	População	População(%)	Area_bairro	Área(%)	Densidade_populacional	preco_metro
1.0	900	900	900	900	900	900	900	900	900	900	900	900
2.0	1172	1172	1172	1172	1172	1172	1172	1172	1172	1172	1172	1172
3.0	360	360	360	360	360	360	360	360	360	360	360	360
4.0	171	171	171	171	171	171	171	171	171	171	171	171
5.0	32	32	32	32	32	32	32	32	32	32	32	32
6.0	5	5	5	5	5	5	5	5	5	5	5	5
8.0	1	1	1	1	1	1	1	1	1	1	1	1
11.0	1	1	1	1	1	1	1	1	1	1	1	1

Pelo boxplot, apenas os apartamentos com menos de quatro vagas não são classificados como outliers. Como há muitos registros com quatro e cinco vagas, não podemos desprezar os apartamentos com menos de 6 vagas.

Já com 6 ou mais vagas temos sete registros, vamos analisá-los:

Figura 23 - registros com 6 ou mais vagas

```
In [66]: #registros com 6 ou mais vagas
         Dados.loc[Dados['vagas'] >= 6]
```

Out[66]:

	preco	area	quartos	banheiros	vagas	suites	bairro	População	População(%)	Area_bairro	Área(%)	Densidade_populacional	preco_metro
795	2200000	240.0	2.0	4.0	6.0	2.0	agua-verde	51461	2.9	4850	1.1	10.6	9166.666667
1089	545000	109.0	3.0	2.0	11.0	1.0	agua-verde	51461	2.9	4850	1.1	10.6	5000.000000
1229	5800000	806.0	4.0	1.0	6.0	4.0	agua-verde	51461	2.9	4850	1.1	10.6	7196.029777
2207	9000000	800.0	4.0	2.0	8.0	4.0	ecoville	25493	1.5	8007	1.8	3.2	11250.000000
2276	6000000	801.0	4.0	5.0	6.0	4.0	ecoville	25493	1.5	8007	1.8	3.2	7490.636704
3062	1580000	350.0	4.0	4.0	6.0	4.0	batel	39234	2.2	4633	1.1	8.5	4514.285714
3646	3500000	600.0	4.0	7.0	6.0	4.0	cabral	24556	1.4	3930	0.9	6.2	5833.333333

Como são apenas sete registros vamos removê-los da base para reduzir as distorções.

Figura 24 - Remoção dos registros com 6 ou mais vagas

```
In [67]: #removendo os registros com mais de 6 vagas
         Dados = Dados.loc[Dados['vagas'] < 6]

In [68]: # variáveis vagas
         Dados.groupby(['vagas']).count()
```

Out[68]:

vagas	preco	area	quartos	banheiros	suites	bairro	População	População(%)	Area_bairro	Área(%)	Densidade_populacional	preco_metro
1.0	900	900	900	900	900	900	900	900	900	900	900	900
2.0	1172	1172	1172	1172	1172	1172	1172	1172	1172	1172	1172	1172
3.0	360	360	360	360	360	360	360	360	360	360	360	360
4.0	171	171	171	171	171	171	171	171	171	171	171	171
5.0	32	32	32	32	32	32	32	32	32	32	32	32

d) Suítes:

Agora vamos agrupar a variável “suites” e obter a contagem dos valores:

Figura 25 - Contagem de apartamentos pelo número de suítes

```
In [69]: # variáveis suites
         Dados.groupby(['suites']).count()
```

Out[69]:

suites	preco	area	quartos	banheiros	vagas	bairro	População	População(%)	Area_bairro	Área(%)	Densidade_populacional	preco_metro
1.0	1861	1861	1861	1861	1861	1861	1861	1861	1861	1861	1861	1861
2.0	344	344	344	344	344	344	344	344	344	344	344	344
3.0	304	304	304	304	304	304	304	304	304	304	304	304
4.0	124	124	124	124	124	124	124	124	124	124	124	124
5.0	2	2	2	2	2	2	2	2	2	2	2	2

Pelo boxplot, apenas os apartamentos com menos de quatro suítes não são classificados como outliers. Como há muitos registros com quatro suítes, não podemos desprezar os apartamentos com menos de 5 suítes.

Já com 5 ou mais suítes temos dois registros, vamos analisá-los:

Figura 26 - registros com 5 ou mais suites

```
In [71]: #registros com mais de 4 suites
         Dados.loc[Dados['suites'] >= 5]
```

Out[71]:

	preco	area	quartos	banheiros	vagas	suites	bairro	População	População(%)	Area_bairro	Área(%)	Densidade_populacional	preco_metro
2672	3800000	390.98	5.0	6.0	3.0	5.0	ecoville	25493	1.5	8007	1.8	3.2	9719.167221
3298	1600000	296.00	5.0	6.0	3.0	5.0	batel	39234	2.2	4633	1.1	8.5	5405.405405

Como são apenas dois registros vamos removê-los da base para reduzir as distorções.

Figura 27 - Remoção dos registros com 5 ou mais vagas

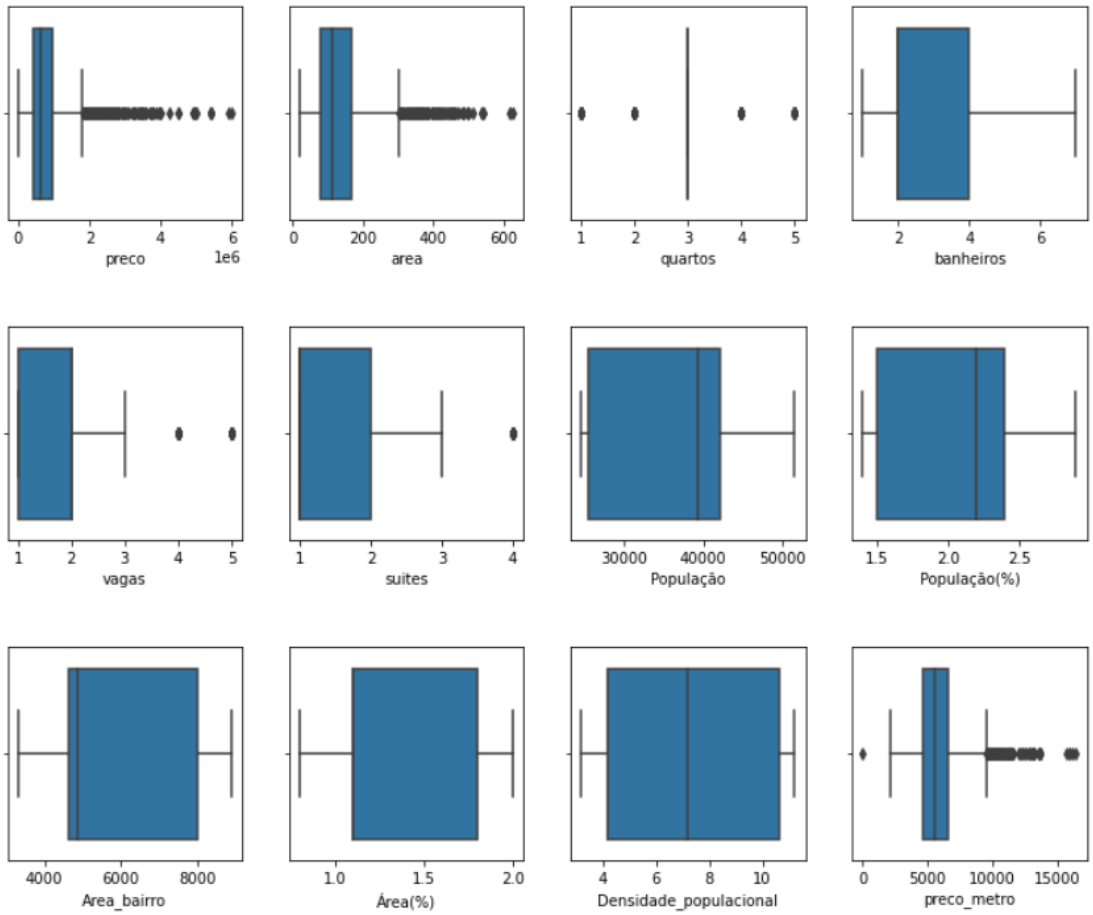
```
In [72]: #removendo os registros com mais de 5 suites
         Dados = Dados.loc[Dados['suites'] < 5]

In [73]: # variáveis suites
         Dados.groupby(['suites']).count()
```

Out[73]:

suites	preco	area	quartos	banheiros	vagas	bairro	População	População(%)	Area_bairro	Área(%)	Densidade_populacional	preco_metro
1.0	1861	1861	1861	1861	1861	1861	1861	1861	1861	1861	1861	1861
2.0	344	344	344	344	344	344	344	344	344	344	344	344
3.0	304	304	304	304	304	304	304	304	304	304	304	304
4.0	124	124	124	124	124	124	124	124	124	124	124	124

Após a remoção de 19 registros poderemos visualizar novamente os boxplots e analisar os outliers.

Figura 28 - Boxplot atributos do dataframe Dados após remoção de outliers



Verifica-se acima que houve uma redução substancial dos outliers das variáveis tratadas, sendo que os outliers que ainda permanecem no dataset não podem ser removidos sob pena de empobrecer demasiadamente a base de dados.

Com relação à variável “área” ainda há muitos outliers mas não será feito nenhum tratamento para não distorcer a base de dados. A variável “preco_metro” é consequência das variáveis “preco” e “area” e não será tratado diretamente.

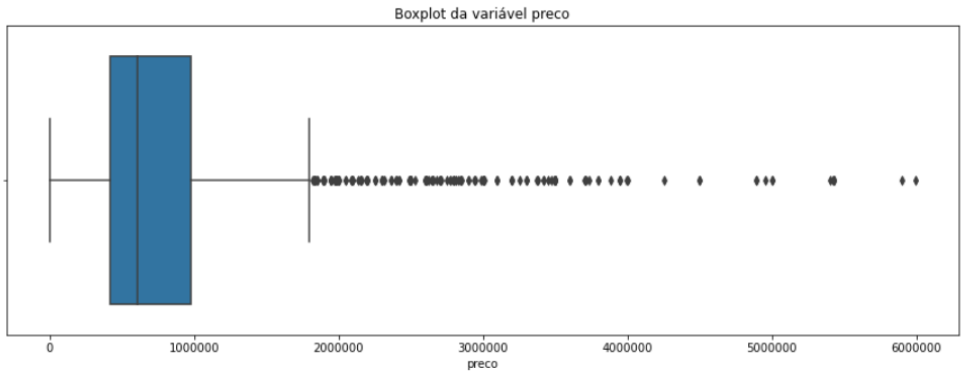
Já a variável “preco” ainda apresenta grande quantidade de outlier, impactando a variável “preco_metro”. Para a variável “preco” apresentaremos uma forma alternativa para tratamento de outliers.

e) Preço:

Vamos na sequência visualizar de forma ampliada o boxplot da variável “preco”:

Figura 29 - Boxplot da variável "preco"

```
In [83]: plt.figure(figsize = (15, 5))
         sns.boxplot(x='preco',data = Dados, orient='h')
         plt.title("Boxplot da variável preco")
         plt.ticklabel_format(axis='x', style='plain' )
```

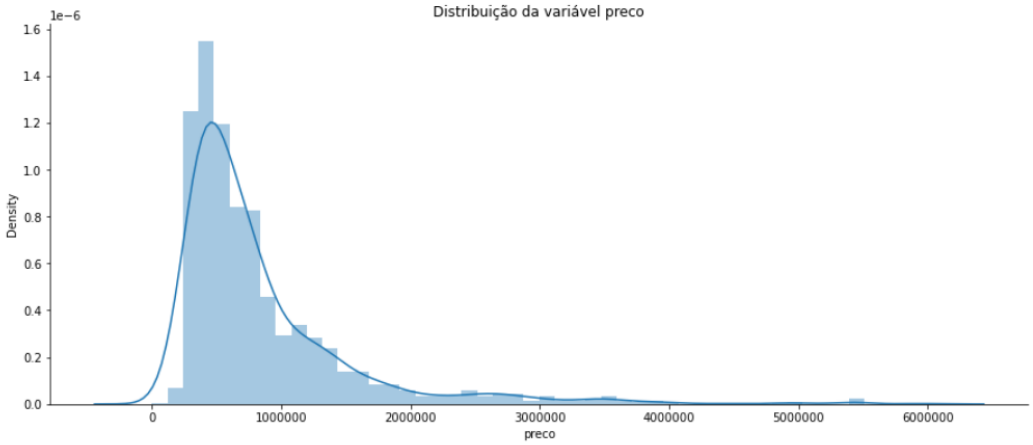


Percebe-se a concentração de registros até próximo de R$ 200 mil por unidade. A partir desse valor temos muitos outliers que precisam ser tratados.

Abaixo temos a distribuição da variável visto em um gráfico de densidade no qual podemos perceber a cauda longa à direita onde estão situados os outliers.

Figura 30 - Gráfico de densidade da variável preço

```
In [84]: # Distribuição da variável 'preco'
sns.distplot(Dados['preco'])
plt.title("Distribuição da variável preco")
plt.ticklabel_format(axis='x', style='plain' )
sns.despine()
```



Para tratar esse outliers vamos utilizar uma abordagem diferenciada. Não vamos substituí-los com valores de mediana, média etc., vamos substituí-los com os valores preditos obtidos por meio de um modelo preliminar.

Basicamente, vamos seguir os seguintes passos:

- cálculo dos limites inferiores e superiores dos outliers por meio do IQR (distância interquartílica);

- criação de um modelo preliminar com a biblioteca Statsmodel;

- criação de uma coluna com os preços preditos pelo modelo preliminar e outra para receber os valores originais, no caso de não outlier, e valores preditos, no caso de que o original seja outlier;

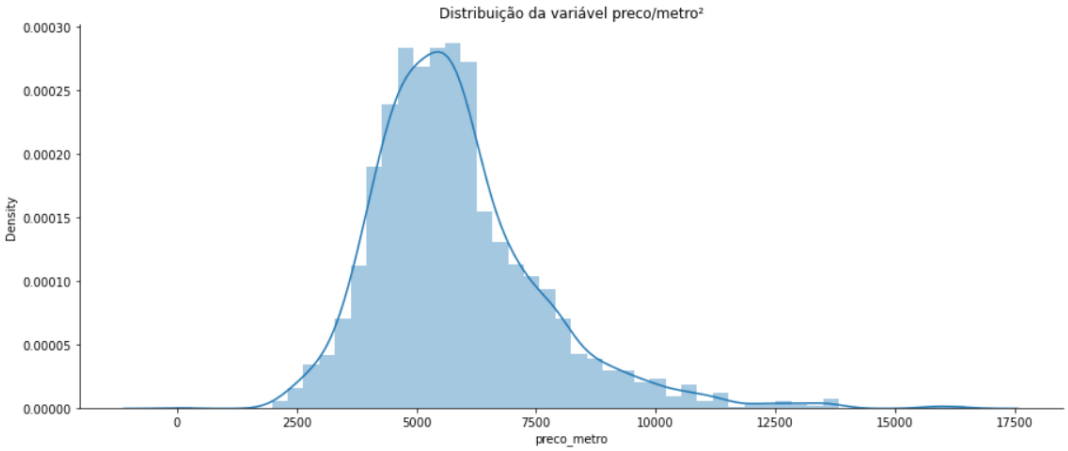
- criação de uma coluna com um marcador de outlier (sim ou não); e

- criação de uma coluna com preços sem outliers (substituição dos preços dos registros marcados com outlier “sim” pelos preços preditos).

Mas antes de seguir estes passos vamos verificar a variável “preco_metro”. Observando o gráfico abaixo vemos que os valores de preços/m² estão concentrados acima de R$ 2.000,00/m².

Figura 31 - Distribuição da variável "preco_metro"

```
In [43]: # Distribuição da variável 'preco_metro'
sns.distplot(Dados['preco_metro'])
plt.title("Distribuição da variável preco/metro²")
plt.ticklabel_format(axis='x', style='plain' )
sns.despine()
```



Filtrando os valores abaixo de R$ 2.000,00/m² temos apenas um registro.

Figura 32 - Valor abaixo de R$ 2.000,00/m²

```
In [44]: Dados.loc[Dados['preco_metro'] < 2000]
```

Out[44]:

	preco	area	quartos	banheiros	vagas	suites	bairro	População	População(%)	Area_bairro	Área(%)	Densidade_populacional	preco_metro
3266	750	145.0	3.0	5.0	2.0	3.0	batel	39234	2.2	4633	1.1	8.5	5.172414

Percebe-se que houve um equívoco no preenchimento do anúncio, sendo o valor do preço do apartamento registrado por R$ 750,00. Estimamos que, dado os demais atributos, o valor correto seria R$ 750 mil e aí procedemos à substituição desse valor antes de continuar o tratamento dos outliers.

Também realizamos a atualização da coluna “preco_metro”. Segue o código para conferência.

Figura 33 - Substituição do valor anunciado incorretamente e atualização da variável "preco_metro"

```
In [45]: #Embora não se trate de outlier, verificamos o erro no anúncio do preço e vamor corrigí-lo para evitar distorçoes
#Substituição do valor do preço de registro em que o valor do preço/metro² era inferior a R$ 2.000,00.
Dados.loc[Dados['preco_metro'] < 2000,'preco']=750000
Dados.loc[Dados['preco_metro'] < 2000,'preco']
```

```
Out[45]: 3266    750000
Name: preco, dtype: int64
```

```
In [46]: #Atualizando a coluna de preço/metro² com valores obtidos a partir da variável "preco_so"
Dados['preco_metro'] = Dados['preco']/Dados['area']
```

Agora sim, seguindo os passos informados mais acima, iniciaremos pelo cálculo da distância interquartílica (IQR)[14]. Por meio da biblioteca Numpy[15] vamos calcular os percentis 25 e 75, os quais são equivalentes ao 1º (Q1) e 3º quartil (Q3), respectivamente. Em seguida vamos subtrair o 1º quartil do 3º obtendo a distância interquartílica (IQR).

Após isso, vamos calcular os limites superiores e inferiores para a detecção de outliers. O limite inferior é obtido subtraindo do Q1 o valor de 1,5 vezes o valor de IQR. Já o limite superior é obtido adicionando ao Q3 o valor de 1,5 vezes o valor de IQR.

Os registros com preços de apartamentos inferiores ao limite inferior ou superior ao limite superior serão considerados outliers e receberão o adequado tratamento. Abaixo podemos observar o código para cálculo dos limites.

Figura 34 - Cálculo dos limites para detecção de outliers

```
In [43]: # Cálculo de Q1, Q3 e IQR:
         Q1 = np.percentile(Dados['preco'], 25)
         Q3 = np.percentile(Dados['preco'], 75)
         IQR = Q3 - Q1
         print(f"IQR: {IQR}")

         # Cálculo dos limites inferiores e superiores para detecção de outliers:
         limite_inferior_outliers = Q1 - 1.5*IQR
         limite_superior_outliers = Q3 + 1.5*IQR
         print(f"Limite inferior para outlier: {limite_inferior_outliers}; Limite superior para outliers: {limite_superior_outliers}")

         IQR: 560000.0
         Limite inferior para outlier: -420000.0; Limite superior para outliers: 1820000.0
```

Na sequência faremos a criação do modelo preliminar, utilizando a biblioteca Statsmodel e o método de regressão linear de mínimos quadrados (OLS – Ordinary Least Squares)[16]. Para o modelo preliminar foram utilizadas apenas as variáveis mais relevantes, considerando para isso o Valor P (P value) abaixo de 0,05 e eliminando as variáveis menos relevante seguindo o processo conhecido como *Stepwise*[17].

Desse processo, que teve como Coeficiente de Determinação (R^2)[18] o valor 0.932, resultaram como variáveis relevantes as seguintes: “area”, “vagas”, “suites”,

[14] Fonte: Curso de Ciência de Dados com Python ministrado pelo Prof. Nélio Machado em 10 e 11/2020 (https://github.com/Alencar2208/DSWP/blob/master/Notebooks/NB15_02__Regress%C3%A3o%20Linear.ipynb)
[15] Disponível em NumPy
[16] Dsiponível em Ordinary least squares - Wikipedia e statsmodels.regression.linear_model.OLS — statsmodels
[17] Disponível em https://towardsdatascience.com/ridge-lasso-and-elasticnet-regression-b1f9c00ea3a3
[18] Disponível em https://pt.wikipedia.org/wiki/Coeficiente_de_determina%C3%A7%C3%A3o#:~:text=O%20coeficiente%20de%20determina%C3%A7%C3%A3o%2C%20tamb%C3%A9m,observados%20de%20uma%20vari%C3%A1vel%20aleat%C3%B3ria

“População”, “Area_bairro”, “Densidade_populacional”, “preco_metro”. Abaixo temos o código desenvolvido.

Figura 35 - Criação e ajuste do modelo preliminar

```
In [44]: Dados.columns

Out[44]: Index(['preco', 'area', 'quartos', 'banheiros', 'vagas', 'suites', 'bairro',
                'População', 'População(%)', 'Area_bairro', 'Área(%)',
                'Densidade_populacional', 'preco_metro'],
               dtype='object')
```

```
In [199]: #Criação do modelo preliminar, utilizando a biblioteca statsmodel
          #(apenas com os atributos mais relevantes => Valor P < 0,05)
          modelo_preliminar = sm.ols(formula = 'preco ~ area + vagas + suites + População\
                                     + Area_bairro + Densidade_populacional + preco_metro', data = Dados)
          modelo_preliminar_treinado = modelo_preliminar.fit()
          modelo_preliminar_treinado.summary()
```

Abaixo podemos ver o sumário da modelo preliminar.

Figura 36 - Resultado do modelo preliminar

Out[50]:

OLS Regression Results

Dep. Variable:	preco	**R-squared:**	0.932
Model:	OLS	**Adj. R-squared:**	0.932
Method:	Least Squares	**F-statistic:**	5168.
Date:	Thu, 22 Apr 2021	**Prob (F-statistic):**	0.00
Time:	12:00:16	**Log-Likelihood:**	-35678.
No. Observations:	2633	**AIC:**	7.137e+04
Df Residuals:	2625	**BIC:**	7.142e+04
Df Model:	7		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-7.622e+05	4.89e+04	-15.583	0.000	-8.58e+05	-6.66e+05
area	6450.4122	81.555	79.093	0.000	6290.493	6610.331
vagas	-4.761e+04	7335.016	-6.490	0.000	-6.2e+04	-3.32e+04
suites	3.853e+04	6616.176	5.824	0.000	2.56e+04	5.15e+04
População	10.8695	1.271	8.551	0.000	8.377	13.362
Area_bairro	-55.0762	6.974	-7.898	0.000	-68.751	-41.401
Densidade_populacional	-6.52e+04	6915.436	-9.429	0.000	-7.88e+04	-5.16e+04
preco_metro	193.3217	2.547	75.902	0.000	188.327	198.316

Omnibus:	1172.297	**Durbin-Watson:**	1.939
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	48832.377
Skew:	1.397	**Prob(JB):**	0.00
Kurtosis:	23.912	**Cond. No.**	5.34e+05

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 5.34e+05. This might indicate that there are
strong multicollinearity or other numerical problems.

Continuando, temos a criação de uma coluna com os preços preditos pelo modelo preliminar e outra coluna que receberá parte dos valores originais (não outliers) e parte dos valores preditos (outliers). A primeira coluna será nomeada como “preco_ajustado_mod_pre” e a segunda como “preco-so” (preço sem outlier). Segue o código.

Figura 37 - Criação de colunas para receber preços preditos e preços sem outliers

```
In [ ]: #Criação da coluna com dados de preços previstos pelo modelo preliminar
        Dados['preco_ajustado_mod_pre']=modelo_preliminar_treinado.fittedvalues

In [ ]: #Criação da coluna com dados de preços ajustados pelo modelo preliminar
        Dados['preco_so']=Dados['preco']
```

No próximo bloco de código vamos criar a coluna para controle de outliers, primeiramente preenchendo todos os registros com a *string* “não”, e, na sequência, substituindo os registros superiores ao limite superior ou inferiores ao limite inferior pela string “sim”. Dessa forma, todos os registros que contém *outliers* no atributo “preco” terão o valor “sim” na coluna “outlier”. Seguem os códigos:

Figura 38 - Criação da coluna para controle de outliers

```
In [ ]: #criação da coluna para controle de outliers
        Dados['outlier']='não'
        Dados.loc[Dados['preco_so']>limite_superior_outliers,'outlier']='sim'
        Dados.loc[Dados['preco_so']<limite_inferior_outliers,'outlier']='sim'
```

E por fim, faremos a substituição na coluna “preco_so” (sem outlier) dos valores de preços com a correspondente marcação na coluna “outlier” com a *string* “sim” pelos valores constantes, dos mesmos registros, na coluna “preco_ajustado_mod_pre”, conforme abaixo.

Figura 39 - Substituição dos outliers na coluna "preco_so"

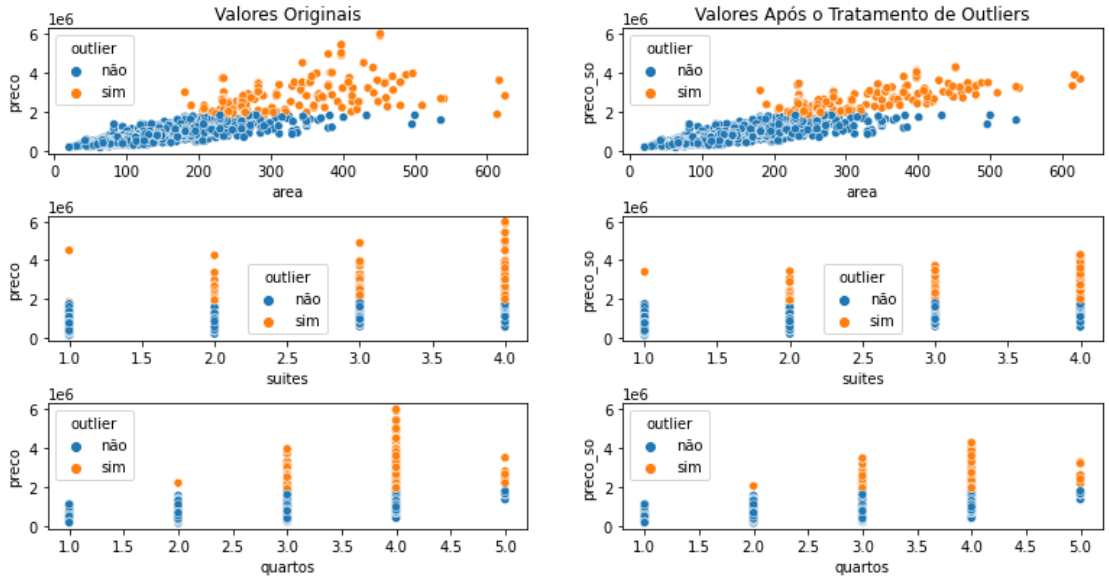
```
In [54]: #Substituição dos outliers na coluna preco_so (sem outlier) pelos valores ajustados do modelo preliminar
         Dados.loc[Dados['outlier']=='sim','preco_so']=Dados.loc[Dados['outlier']=='sim']['preco_ajustado_mod_pre']
         Dados.loc[Dados['outlier']=='sim'].head()
```

Out[54]:

s	vagas	suites	bairro	População	População(%)	Area_bairro	Área(%)	Densidade_populacional	preco_metro	preco_ajustado_mod_pre	preco_so	outlier
)	3.0	2.0	portao	42038	2.4	5838	1.3	7.2	6385.977273	2.059291e+06	2.059291e+06	sim
)	4.0	2.0	agua-verde	51461	2.9	4850	1.1	10.6	16459.399823	3.076850e+06	3.076850e+06	sim
)	3.0	4.0	agua-verde	51461	2.9	4850	1.1	10.6	7243.816254	2.076088e+06	2.076088e+06	sim
)	4.0	4.0	agua-verde	51461	2.9	4850	1.1	10.6	9227.022118	2.575332e+06	2.575332e+06	sim
)	4.0	2.0	agua-verde	51461	2.9	4850	1.1	10.6	9615.384615	3.435505e+06	3.435505e+06	sim

No gráfico abaixo podemos visualizar a alteração de preço antes e depois do tratamento de outliers. Ressalta-se que a coluna “preco” mantem os valores originais e a coluna “preco_so” possui os dados tratados.

Figura 40 - Valores de preço antes e depois do tratamento de outliers



Como se pode perceber acima, após o tratamento de outliers ocorreu uma menor dispersão dos valores, ficando mais concentrados próximo de uma linha de tendência. O código para construção do gráfico acima pode ser verificado abaixo:

Figura 41 - Código para construção dos gráficos visualizados anteriormente

```
In [201]: plt.figure()
          ax1 = plt.subplot(321)
          sns.scatterplot(y=Dados.preco, x=Dados.area, hue = Dados.outlier)
          plt.title('Valores Originais')


          ax2 = plt.subplot(322, sharey=ax1)
          sns.scatterplot(y=Dados.preco_so, x=Dados.area, hue = Dados.outlier)
          plt.title('Valores Após o Tratamento de Outliers')


          ax3 = plt.subplot(323)
          sns.scatterplot(y=Dados.preco, x=Dados.suites, hue = Dados.outlier)


          ax4 = plt.subplot(324, sharey=ax3)
          sns.scatterplot(y=Dados.preco_so, x=Dados.suites, hue = Dados.outlier)

          ax5 = plt.subplot(325)
          sns.scatterplot(y=Dados.preco, x=Dados.quartos, hue = Dados.outlier)

          ax6 = plt.subplot(326, sharey=ax5)
          sns.scatterplot(y=Dados.preco_so, x=Dados.quartos, hue = Dados.outlier)

          plt.subplots_adjust(bottom=0.1, right=0.8, top=0.9, hspace=0.5)

          plt.show()
```

Agora vamos verificar na construção de um segundo modelo preliminar se houve uma melhora no R^2 após o tratamento de outliers na variável “preco” (no caso foi utilizado a variável “preco_so”, a qual contém os preços sem outliers).

Figura 42 - Criação do 2° modelo preliminar

```
In [48]: #Criação do 2º modelo preliminar, utilizando a biblioteca statsmodel (melhoria no R²)
         #(apenas com os atributos mais relevantes => Valor P < 0,05)
         modelo_preliminar_2 = sm.ols(formula = 'preco_so ~ area + vagas + suites + População + \
                                                 Area_bairro + Densidade_populacional + preco_metro', data = Dados)
         modelo_preliminar_treinado_2 = modelo_preliminar_2.fit()
         modelo_preliminar_treinado_2.summary()
```

Figura 43 - Resultado do 2° modelo preliminar

Out[56]:

OLS Regression Results

Dep. Variable:	preco_so	**R-squared:**	0.968
Model:	OLS	**Adj. R-squared:**	0.968
Method:	Least Squares	**F-statistic:**	1.140e+04
Date:	Thu, 22 Apr 2021	**Prob (F-statistic):**	0.00
Time:	12:00:21	**Log-Likelihood:**	-34412.
No. Observations:	2633	**AIC:**	6.884e+04
Df Residuals:	2625	**BIC:**	6.889e+04
Df Model:	7		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-6.856e+05	3.02e+04	-22.673	0.000	-7.45e+05	-6.26e+05
area	5838.6602	50.420	115.800	0.000	5739.792	5937.528
vagas	-1.217e+04	4534.780	-2.683	0.007	-2.11e+04	-3276.108
suites	4.531e+04	4090.366	11.077	0.000	3.73e+04	5.33e+04
População	6.9431	0.786	8.834	0.000	5.402	8.484
Area_bairro	-34.7359	4.311	-8.057	0.000	-43.190	-26.282
Densidade_populacional	-4.332e+04	4275.381	-10.133	0.000	-5.17e+04	-3.49e+04
preco_metro	156.6427	1.575	99.478	0.000	153.555	159.730

Omnibus:	1213.879	**Durbin-Watson:**	1.944
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	20142.180
Skew:	-1.756	**Prob(JB):**	0.00
Kurtosis:	16.087	**Cond. No.**	5.34e+05

Vemos que houve uma melhora do Coeficiente de Determinação (R^2) que antes era 0,932 e agora é 0,968, o que denota que estamos evoluindo na construção do modelo.

Embora não tenha sido realizado o tratamento de *outliers* com outras técnicas como média, mediana etc., entendemos que da forma como foi realizada há uma distorção menor dos dados, pois nesta técnica o peso de outros atributos como quartos, suítes etc., são considerados na hora de calcular os valores que serão utilizados para substituir os outliers.

4. Análise e Exploração dos Dados

Agora vamos analisar e explorar a base de dados em busca de padrões e/ou informações que possam ser úteis para melhor compreensão das relações existentes entre os diversos atributos dos apartamentos e bairros de Curitiba-PR constantes do dataset.

Primeiro, vamos observar as tabelas abaixo com as estatísticas do *dataset* “Dados” fornecidas pelo método *describe()* da biblioteca Pandas.

Tabela 3 - Estatísticas do dataset Dados – parte 1

	preco	area	quartos	banheiros	vagas	suites	População	População(%)	Area_bairro
count	2.633000e+03	2633.000000	2633.000000	2633.000000	2633.000000	2633.000000	2633.000000	2633.000000	2633.000000
mean	8.449052e+05	138.006251	2.984808	2.781618	1.960501	1.502848	37245.368781	2.114926	5761.985188
std	7.137591e+05	81.554500	0.717082	1.378767	0.919621	0.873936	9783.423061	0.535560	1763.752412
min	1.400000e+05	21.000000	1.000000	1.000000	1.000000	1.000000	24556.000000	1.400000	3310.000000
25%	4.200000e+05	80.880000	3.000000	2.000000	1.000000	1.000000	25493.000000	1.500000	4633.000000
50%	6.130000e+05	115.000000	3.000000	2.000000	2.000000	1.000000	39234.000000	2.200000	4850.000000
75%	9.800000e+05	168.830000	3.000000	4.000000	2.000000	2.000000	42038.000000	2.400000	8007.000000
max	5.990000e+06	625.000000	5.000000	7.000000	5.000000	4.000000	51461.000000	2.900000	8896.000000

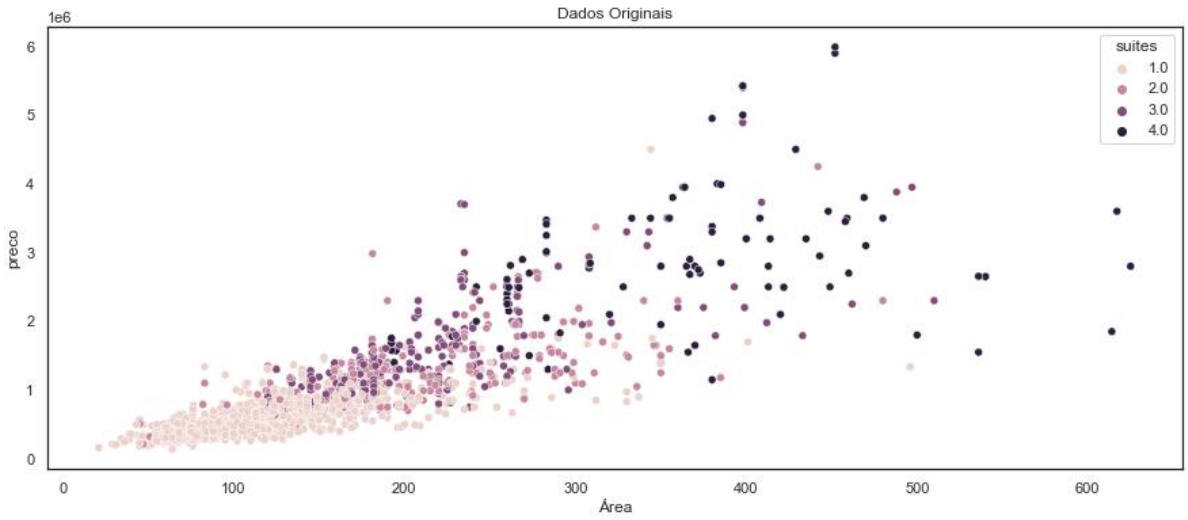
Tabela 4 - Estatísticas do dataset Dados – parte 2

	Área(%)	Densidade_populacional	preco_metro	preco_ajustado_mod_pre	preco_so
count	2633.000000	2633.00000	2633.000000	2.633000e+03	2.633000e+03
mean	1.310330	7.13646	5845.839158	8.449052e+05	8.293821e+05
std	0.383185	2.79391	1789.538744	6.891926e+05	6.431295e+05
min	0.800000	3.20000	2170.542636	-3.369280e+05	1.400000e+05
25%	1.100000	4.20000	4669.117647	3.788078e+05	4.200000e+05
50%	1.100000	7.20000	5569.620253	6.482572e+05	6.130000e+05
75%	1.800000	10.60000	6619.047619	1.100656e+06	9.800000e+05
max	2.000000	11.20000	16459.399823	4.277666e+06	4.277666e+06

Vemos acima que os apartamentos possuem em média 138 m², cerca de 3 quartos, 2,78 banheiros, 1,96 vagas e 1,5 suítes.

A seguir, no gráfico abaixo temos a comparação do preço anunciado com a área do apartamento.

Figura 44 - Preço x Área



Nesse gráfico vemos nitidamente que ao passo que aumenta a área dos apartamentos os preços ficam mais dispersos. Isso pode denotar o incremento no grau de diferenciação dos imóveis em função de outros atributos que não são capturados no *dataset*, como qualidade dos materiais, idade do imóvel, necessidade de efetuar a venda em um curto prazo etc.

Nos apartamentos com área menor vemos uma maior concentração dos preços, significando o menor grau de diferenciação e a maior importância do atributo área para definição do preço. Pode-se inferir que, nessa região do gráfico, os imóveis se comportam como commodities, simplificando a comparação entre as unidades.

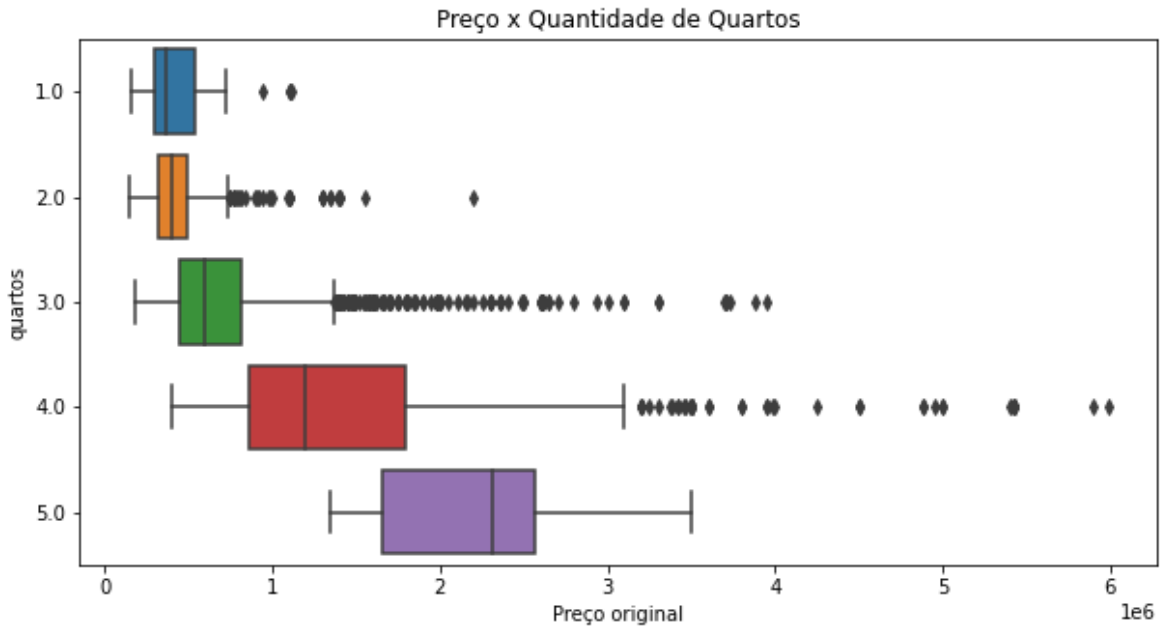
Segue o código para geração do gráfico:

Figura 45 - código para geração do gráfico acima

```
In [100]: #Dispersão dos preços conforme aumenta o tamanho do apartamento.
          plt.figure()
          sns.scatterplot(Dados.area, Dados.preco, hue = Dados.suites)
          plt.title('Dados Originais')
          plt.xlabel('Área')
```

Comparando preços por quartos, podemos ver no gráfico abaixo que a concentração de valores ocorre em maior intensidade nos apartamentos de até três quartos.

Figura 46 - Preços x Quartos



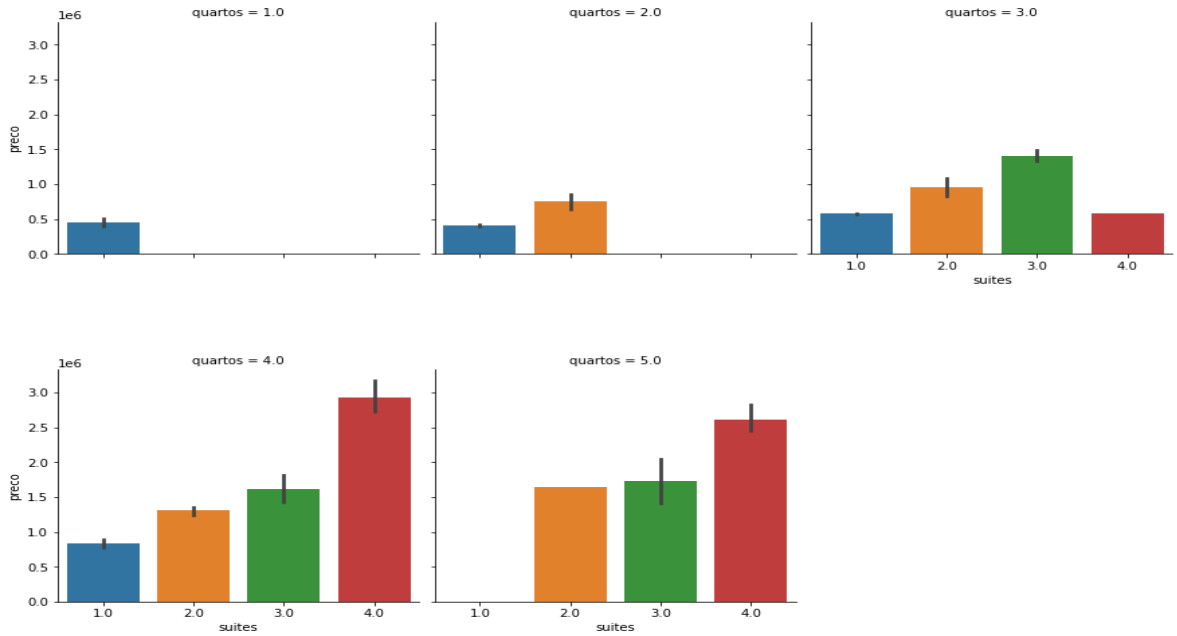
O código para gerar o gráfico acima pode ser visto a seguir.

```
In [59]: #Maior concentração de preços em apartamentos até três quartos
         plt.figure(figsize = (10, 5))
         sns.boxplot(x='preco',y='quartos',data = Dados, orient='h')
         plt.xlabel('Preço original')
         plt.title('Preço x Quantidade de Quartos')
```

Quando comparamos quartos, suítes e preços podemos perceber que a dispersão maior nos valores ocorre nos apartamentos com mais de quatro quartos e três suítes.

Isso pode ser visto no comprimento da linha preta no topo de cada coluna. Quanto maior, mais dispersos são os dados.

Figura 47 - Comparação quartos, suítes e preço



Segue o código para construção deste gráfico:

```
In [60]: #Maior existência de outliers na variável preço para apartamentos com 3 ou mais suítes.
         #Verificar unidades com mais suítes que quartos
         plt.figure()
         sns.catplot(x = 'suites', y='preco',data = Dados, col = 'quartos', col_wrap=3, kind='bar' ,orient='v')
         plt.subplots_adjust(bottom=0.1, right=0.8, top=0.9, hspace=0.5)
```

Percebemos no gráfico acima, mais um erro de anúncio. Na coluna mais acima e à direita temos uma unidade com três quartos e quatro suítes. Como isso pode distorcer os cálculos de regressão linear, vamos efetuar a correção, igualando a quantidade de suítes ao número de quartos. Abaixo o código para correção.

```
In [61]: # Verificando quantidade de suítes maior que quartos (3 quartos => 4 suítes)
         Dados.loc[(Dados['quartos']==3.0) & (Dados['suites']==4.0)]
         #Dados[822:823]
```

Out[61]:

	preco	area	quartos	banheiros	vagas	suites	bairro	População	População(%)	Area_bairro	Área(%)	Densidade_populacional	preco_metro	preco_a
823	575000	110.0	3.0	1.0	1.0	4.0	agua-verde	51461	2.9	4850	1.1	10.6	5227.272727	

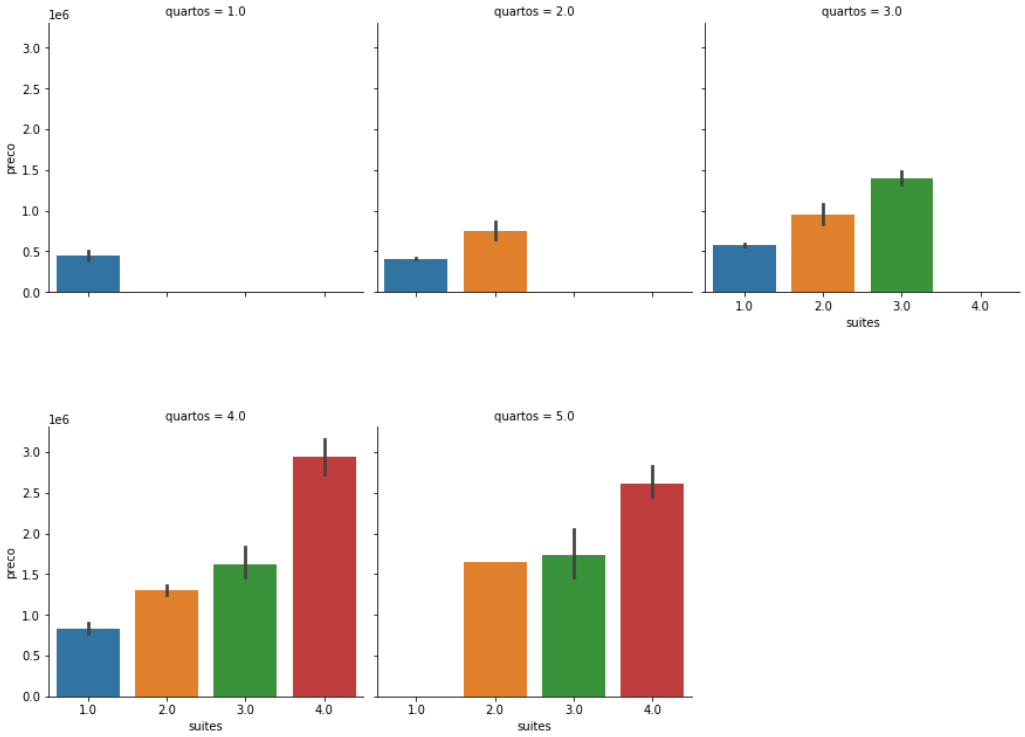
```
In [62]: #Igualando a quantidade de suítes à quantidade de quartos
         Dados.loc[(Dados['quartos']==3.0) & (Dados['suites']==4.0),'suites']=Dados.loc[(Dados['quartos']==3.0) & (Dados['suites']==4.0)][
         Dados.loc[(Dados['quartos']==3.0) & (Dados['suites']==4.0)]
```

Out[62]:

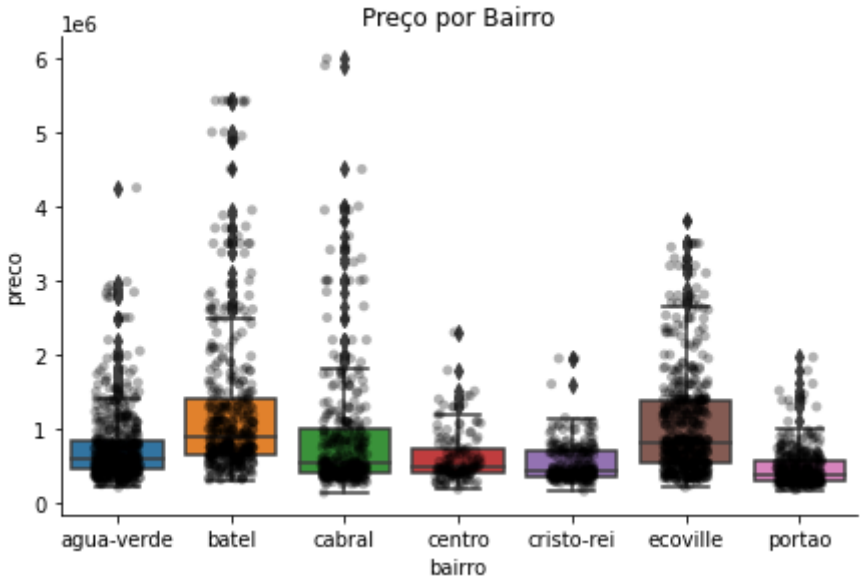
	preco	area	quartos	banheiros	vagas	suites	bairro	População	População(%)	Area_bairro	Área(%)	Densidade_populacional	preco_metro	preco_ajustad

E conferindo novamente o gráfico vemos que o problema foi resolvido.

Figura 48 - Comparativo de quartos, suítes e preço atualizado



Agora, comparando preços por bairros, podemos ver no gráfico abaixo em quais bairros estão localizados aqueles apartamentos diferenciados, ou seja, maiores e com mais quantidades de quartos e suítes.



Na sequência, Cabral, Batel, Água-Verde e Ecoville são os bairros onde estão localizados os apartamentos com maior grau de diferenciação. Centro, Cristo-Rei e

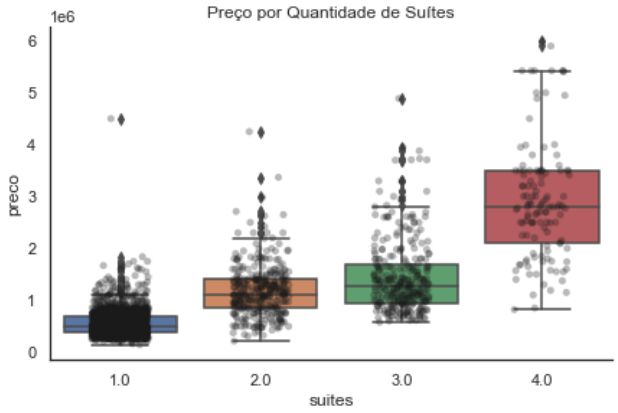
Portão são os bairros onde estão concentrados os apartamentos de menor valor. Segue o código para confecção do gráfico acima.

In [64]:
```
plt.figure()
sns.catplot(x='bairro', y = 'preco', kind = "box", data = Dados, height = 4, aspect = 1.5)
sns.stripplot(x = 'bairro', y = 'preco', data = Dados, alpha = 0.3, jitter = 0.2, color = 'k');
plt.title('Preço por Bairro')
plt.show()
```

Agora com o mesmo gráfico acima (e respectivo código) vamos segregar por suítes, quartos, vagas e banheiros.

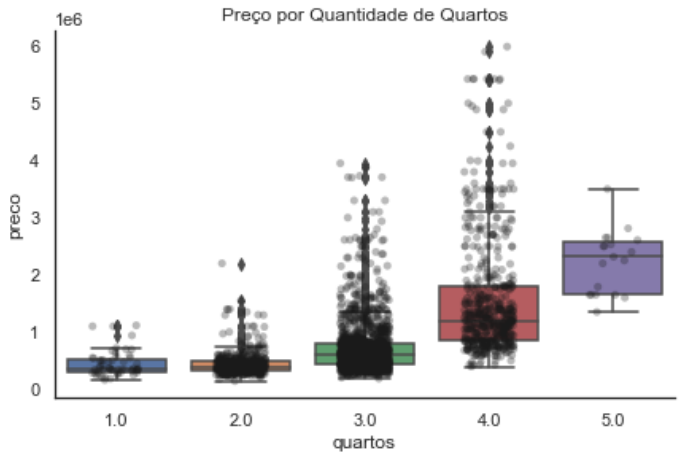
In [172]:
```
plt.figure()
sns.catplot(x='suites', y = 'preco', kind = "box", data = Dados, height = 4, aspect = 1.5)
sns.stripplot(x = 'suites', y = 'preco', data = Dados, alpha = 0.3, jitter = 0.2, color = 'k');
plt.title('Preço por Quantidade de Suítes')
plt.show()
```

```
<Figure size 1080x432 with 0 Axes>
```



:
```
plt.figure()
sns.catplot(x='quartos', y = 'preco', kind = "box", data = Dados, height = 4, aspect = 1.5)
sns.stripplot(x = 'quartos', y = 'preco', data = Dados, alpha = 0.3, jitter = 0.2, color = 'k');
plt.title('Preço por Quantidade de Quartos')
plt.show()
```

```
<Figure size 1080x432 with 0 Axes>
```



Comparando quartos com suítes vemos que, além da dispersão maior de valores para quantidades maiores de quartos ou suítes, com cinco quartos a regra é que praticamente todos sejam suítes.

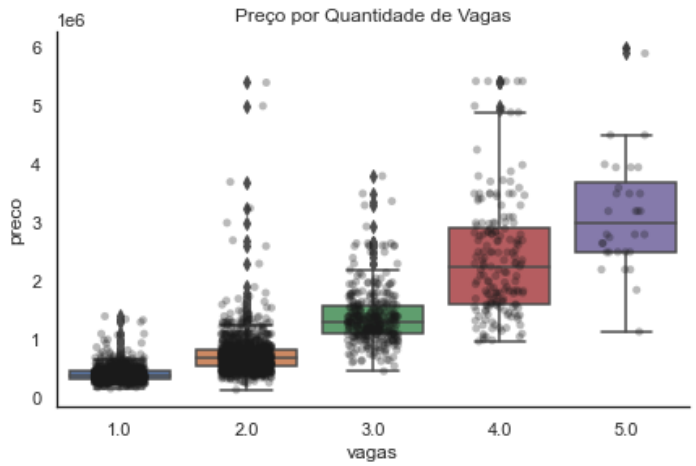
Tabela 5 - Quantidade de quartos x suítes

quartos / suites	1.0	2.0	3.0	4.0	5.0
1.0	54.0	468.0	1193.0	146.0	-
2.0	-	48.0	52.0	240.0	4.0
3.0	-	-	253.0	48.0	4.0
4.0	-	-	-	112.0	11.0

Muita dispersão de valores também é percebido para apartamentos com mais de 3 vagas.

```
plt.figure()
sns.catplot(x='vagas', y = 'preco', kind = "box", data = Dados, height = 4, aspect = 1.5)
sns.stripplot(x = 'vagas', y = 'preco', data = Dados, alpha = 0.3, jitter = 0.2, color = 'k');
plt.title('Preço por Quantidade de Vagas')
plt.show()
```

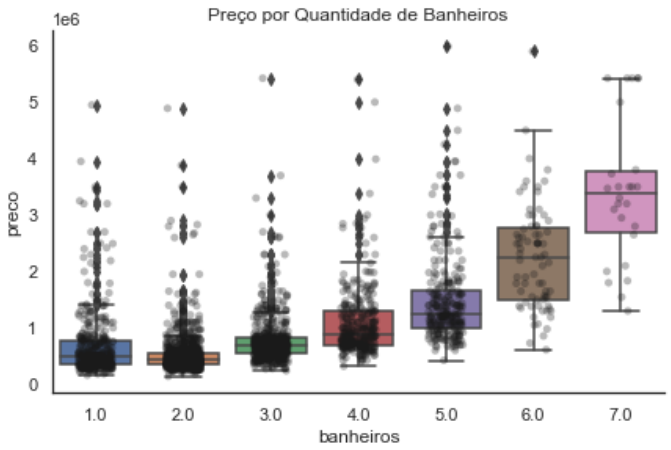
```
<Figure size 1080x432 with 0 Axes>
```



Com relação ao número de banheiros, a dispersão ocorre a partir de quatro banheiros.

```
plt.figure()
sns.catplot(x='banheiros', y = 'preco', kind = "box", data = Dados, height = 4, aspect = 1.5)
sns.stripplot(x = 'banheiros', y = 'preco', data = Dados, alpha = 0.3, jitter = 0.2, color = 'k');
plt.title('Preço por Quantidade de Banheiros')
plt.show()
```

```
<Figure size 1080x432 with 0 Axes>
```



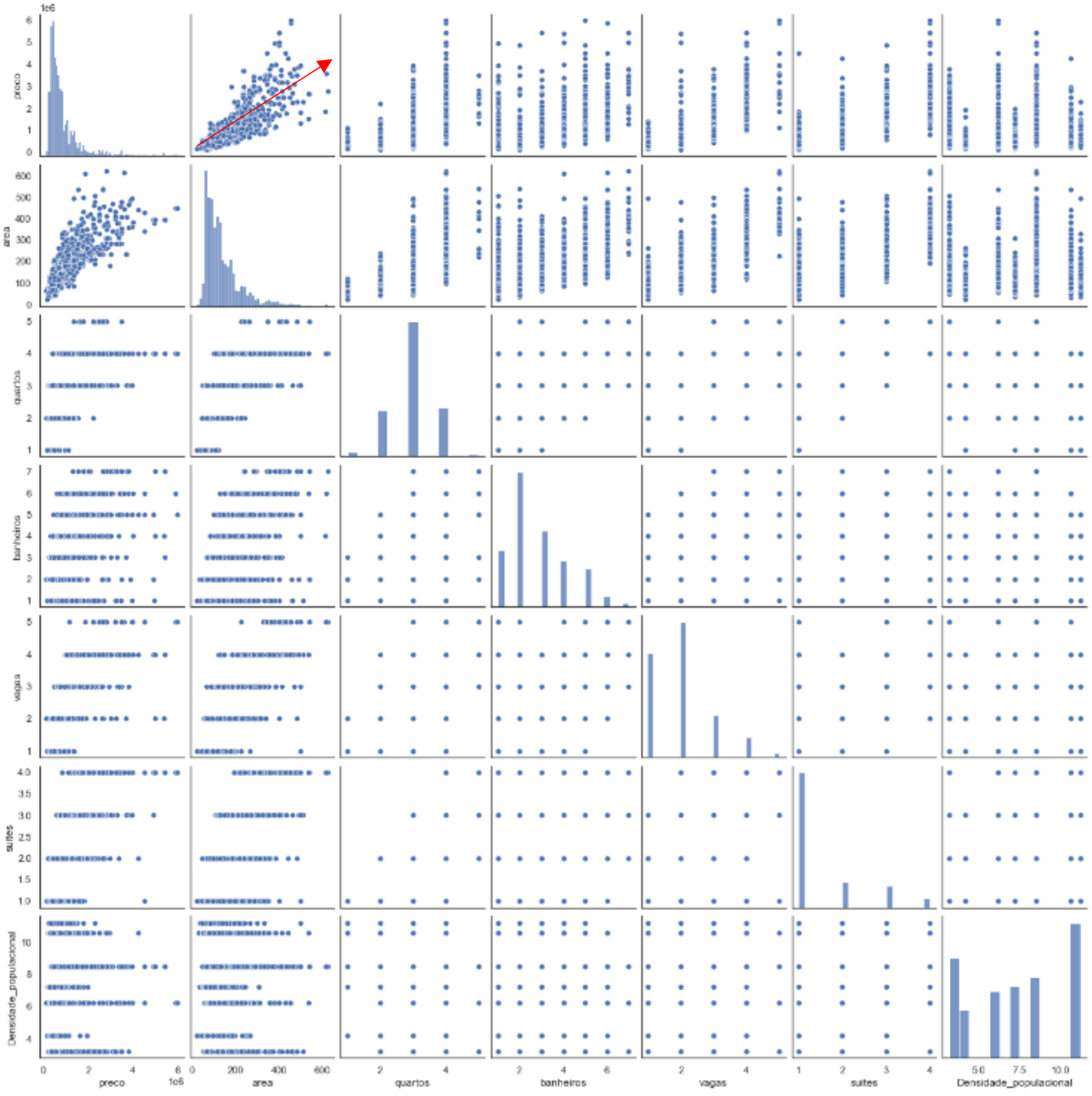
Na tabela abaixo verificamos a contagem de apartamentos por banheiros, quartos e suítes.

Tabela 6 - Comparativo de banheiros, quartos e suítes

quartos	1.0	2.0		3.0			4.0				5.0		
suítes	**1.0**	**1.0**	**2.0**	**1.0**	**2.0**	**3.0**	**1.0**	**2.0**	**3.0**	**4.0**	**2.0**	**3.0**	**4.0**
banheiros													
1.0	34.0	113.0	4.0	171.0	3.0	18.0	13.0	34.0	7.0	11.0	-	-	-
2.0	19.0	331.0	14.0	545.0	7.0	4.0	24.0	17.0	2.0	5.0	1.0	-	1.0
3.0	1.0	22.0	17.0	366.0	25.0	27.0	54.0	27.0	5.0	1.0	3.0	-	-
4.0	-	2.0	10.0	107.0	12.0	62.0	50.0	56.0	11.0	21.0	-	-	-
5.0	-	-	3.0	4.0	5.0	119.0	4.0	95.0	18.0	22.0	-	3.0	1.0
6.0	-	-	-	-	-	20.0	1.0	9.0	5.0	33.0	-	1.0	7.0
7.0	-	-	-	-	-	3.0	-	2.0	-	19.0	-	-	2.0

Pelo *pairplot* abaixo, construído por meio da biblioteca Seaborn[19], podemos perceber como se relacionam algumas variáveis.

Figura 49 - Pairplot - relação entre as variáveis

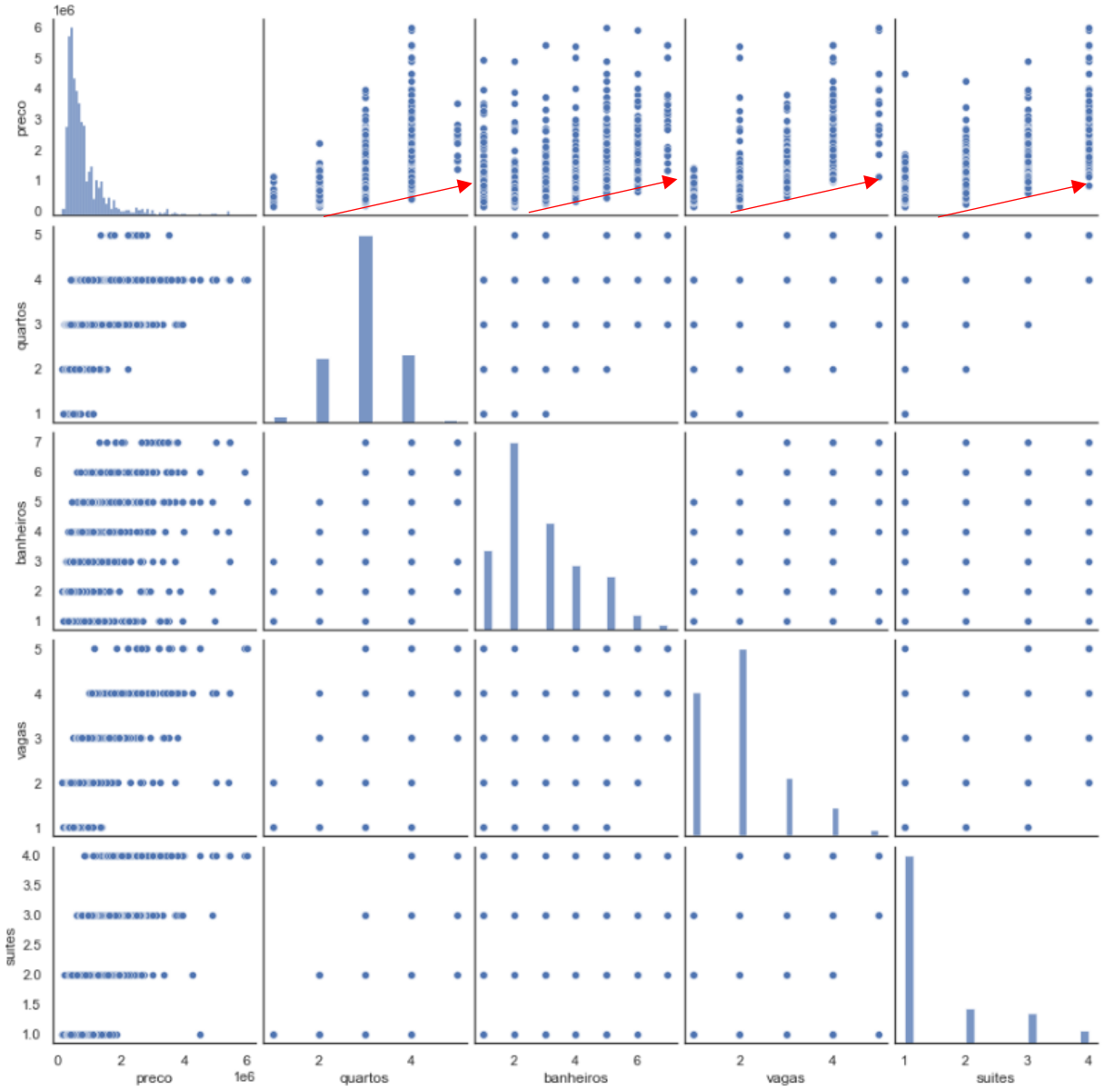


A mais nítida relação, como já mencionado anteriormente, é entre preço e área. Quanto maior a área, maior o preço.

Com relação a quartos, banheiros, vagas e suítes, percebemos que há uma relação direta no preço em sua base, ou seja, há aparentemente um piso a ser respeitado. Vamos ampliar a imagem para melhor visualizar

[19] Disponível em https://seaborn.pydata.org/

Figura 50 - Comparativo preço x quartos, banheiros, vagas e suítes



Segue o código para construção dos pairplots.

```
In [217]: sns.pairplot(Dados[['preco', 'area', 'quartos', 'banheiros', 'vagas', 'suites', 'Densidade_populacional']])
          plt.show()
```

4.1 Análise da correlação entre atributos

Um dos fatores mais importantes para o cálculo de regressão linear é a análise da correlação entre atributos.

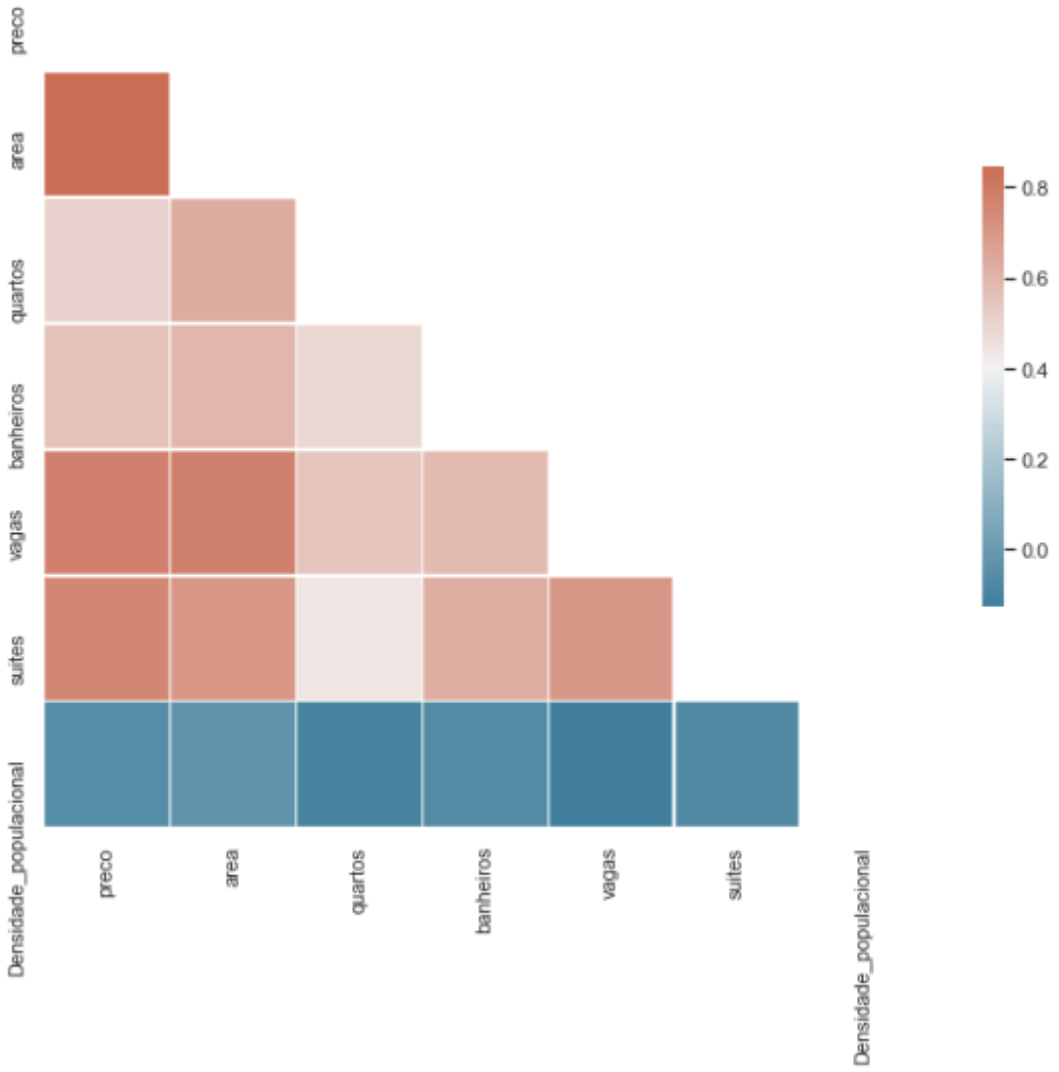
Quando há variáveis correlacionadas, ou seja, quando uma variável aumenta de valor a outra também aumenta, ou, quando uma aumenta de valor a outra diminui,

pode ocorrer de o modelo linear não ajustar seus pesos adequadamente, a depender da força da correlação.

Nesses casos o modelo pode não servir para predizer resultados futuros, ou, no caso em análise, para fazer avaliação de imóveis de acordo com as ofertas no mercado.

Uma ferramenta muito utilizada para visualizar a correlação dos atributos é o mapa de calor abaixo[20] o qual foi obtido por meio da biblioteca Seaborn e adaptado para o nosso caso.

Figura 51 - Heatmap - correlação dos atributos



[20] Disponível em https://seaborn.pydata.org/examples/many_pairwise_correlations.html

Quanto mais próximo de 1 ou de -1, mais forte é a correlação dos atributos. No caso as variáveis mais correlacionadas são preço e área, preço e suítes, vagas e área, suítes e vagas. Abaixo segue o código para a geração do gráfico:

In [221]:
```
#Gráfico das correlações entre as features/variáveis/colunas
#Source: https://seaborn.pydata.org/examples/many_pairwise_correlations.html
import seaborn as sns
from string import ascii_letters
import matplotlib.pyplot as plt

sns.set_theme(style = "white")

d = Dados[['preco', 'area', 'quartos', 'banheiros', 'vagas', 'suites', 'Densidade_populacional']]

# Compute the correlation matrix
corr = d.corr()

# Generate a mask for the upper triangle
mask = np.triu(np.ones_like(corr, dtype=bool))

# Set up the matplotlib figure
f, ax = plt.subplots(figsize=(11, 9))

# Generate a custom diverging colormap
cmap = sns.diverging_palette(230, 20, as_cmap=True)

# Draw the heatmap with the mask and correct aspect ratio
sns.heatmap(corr, mask=mask, cmap=cmap, center=0.4,
            square=True, linewidths=.5, cbar_kws={"shrink": .5})
```

A tabela abaixo apresenta os valores de correlação dos atributos. Estima-se que as variáveis mais correlacionadas com a variável preço possam servir melhor para explicar o modelo.

In [222]:
```
d.corr()
```

Out[222]:

	preco	area	quartos	banheiros	vagas	suites	Densidade_populacional
preco	1.000000	0.846866	0.506918	0.564747	0.781479	0.755345	-0.058115
area	0.846866	1.000000	0.630357	0.594166	0.777294	0.708096	-0.027909
quartos	0.506918	0.630357	1.000000	0.487376	0.551618	0.445504	-0.102585
banheiros	0.564747	0.594166	0.487376	1.000000	0.585902	0.628074	-0.064892
vagas	0.781479	0.777294	0.551618	0.585902	1.000000	0.700004	-0.128179
suites	0.755345	0.708096	0.445504	0.628074	0.700004	1.000000	-0.067247
Densidade_populacional	-0.058115	-0.027909	-0.102585	-0.064892	-0.128179	-0.067247	1.000000

4.2 Problema da dupla contagem de quartos e suítes

Comumente, em casos envolvendo imóveis, tem-se observado um problema de dupla contagem de quartos e suítes, bem como de banheiros e suítes.

Este problema, em tese, tenderia a prejudicar o desempenho dos modelos de previsão ao alocar de parte do valor do imóvel ao quarto e parte à suíte, sendo que muitas vezes pode se tratar do mesmo espaço físico.

Ao tentar superar esse problema, vamos experimentar uma alternativa que considera um conceito diferente para quarto. Nessa abordagem quarto é conceituado como um quarto que não é uma suíte. Assim quarto será definido como um quarto sem banheiro e suíte como um quarto com banheiro.

Definido o novo conceito de quarto, agora precisamos ajustar os valores da variável “quartos”. Vamos considerar que a quantidade de suítes está duplamente contada na variável “quartos”.

Desse modo procederemos à subtração da variável “quartos” pela quantidade de suítes, obtendo a quantidade de quartos sem banheiros.

Abaixo vemos um resumo da quantidade de quartos comparado com suítes:

```
In [301]: # Comparando, quantidade de suítes e quartos
          Dados.pivot_table(values='preco', index=['quartos'], columns=['suites'], aggfunc=np.count_nonzero)
```

Out[301]:

quartos \ suites	1.0	2.0	3.0	4.0
1.0	54.0	NaN	NaN	NaN
2.0	468.0	48.0	NaN	NaN
3.0	1193.0	52.0	253.0	NaN
4.0	146.0	240.0	48.0	112.0
5.0	NaN	4.0	4.0	11.0

Na sequência fazemos a subtração mencionada acima, obtendo a quantidade de quartos sem suítes.

```
In [302]: #ajustando o dataset para retirar o número de suítes do número de quartos em função da contagem em duplicidade
          Dados['quartos']=Dados['quartos']-Dados['suites']
          Dados.head()
```

Out[302]:

	preco	area	quartos	banheiros	vagas	suites	bairro	População	População(%)	Area_bairro	Área(%)	Densidade_populacional	preco_metro	preco_aju
0	286000	63.00	1.0	2.0	1.0	1.0	portao	42038	2.4	5838	1.3	7.2	4539.682540	
1	328000	75.00	2.0	2.0	1.0	1.0	portao	42038	2.4	5838	1.3	7.2	4373.333333	
2	370000	62.77	1.0	2.0	2.0	1.0	portao	42038	2.4	5838	1.3	7.2	5894.535606	
3	295000	62.88	1.0	2.0	1.0	1.0	portao	42038	2.4	5838	1.3	7.2	4691.475827	
4	260000	75.00	2.0	2.0	1.0	1.0	portao	42038	2.4	5838	1.3	7.2	3466.666667	

E obtendo novamente o resumo:

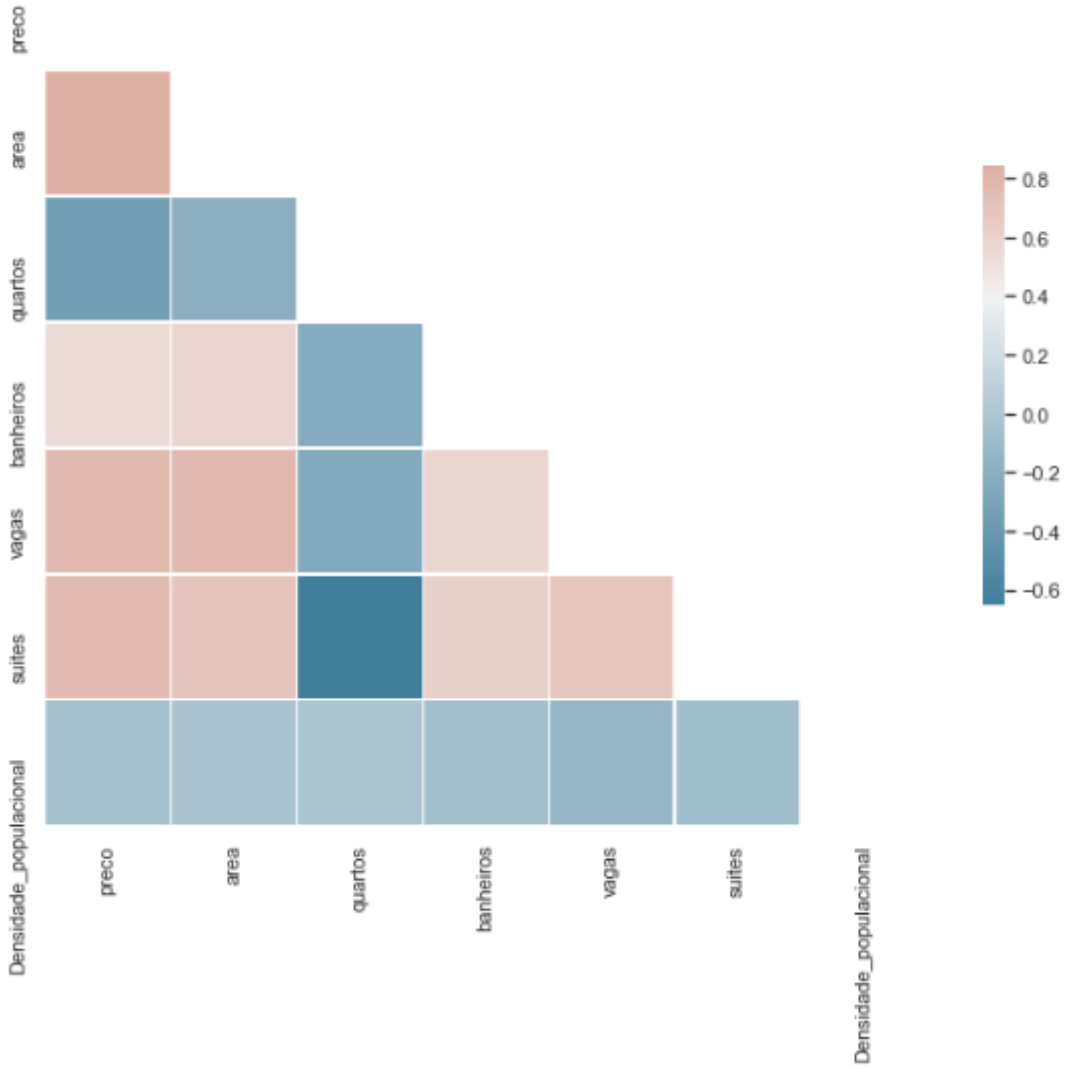
```
In [303]: # Comparando, quantidade de suítes e quartos
          Dados.pivot_table(values='preco', index=['quartos'], columns=['suites'], aggfunc=np.count_nonzero)
```

Out[303]:

quartos \ suites	1.0	2.0	3.0	4.0
0.0	54.0	48.0	253.0	112.0
1.0	468.0	52.0	48.0	11.0
2.0	1193.0	240.0	4.0	NaN
3.0	146.0	4.0	NaN	NaN

Observando novamente o mapa de calor vemos que as correlações entre preço e quarto, e, entre quarto e suítes, foram afetadas.

<AxesSubplot:>



Vendo os números da tabela abaixo, percebemos que agora quartos e suítes são inversamente correlacionadas, o que nos parece bem lógico, pois nos apartamentos maiores há mais suítes e menos quartos sem banheiro.

```
In [305]: #A correlação entre suítes/quartos é reduzida em função das alterações acima.
          #Agora quartos e suítes são inversamente proporcionais, ou seja, aumentando o número de suítes diminui o de quartos
          d.corr()
```

Out[305]:

	preco	area	quartos	banheiros	vagas	suites	Densidade_populacional
preco	1.000000	0.846866	-0.349132	0.564747	0.781479	0.755345	-0.058115
area	0.846866	1.000000	-0.196053	0.594166	0.777294	0.708096	-0.027909
quartos	-0.349132	-0.196053	1.000000	-0.234585	-0.254324	-0.653052	-0.017518
banheiros	0.564747	0.594166	-0.234585	1.000000	0.585902	0.628074	-0.064892
vagas	0.781479	0.777294	-0.254324	0.585902	1.000000	0.700004	-0.128179
suites	0.755345	0.708096	-0.653052	0.628074	0.700004	1.000000	-0.067247
Densidade_populacional	-0.058115	-0.027909	-0.017518	-0.064892	-0.128179	-0.067247	1.000000

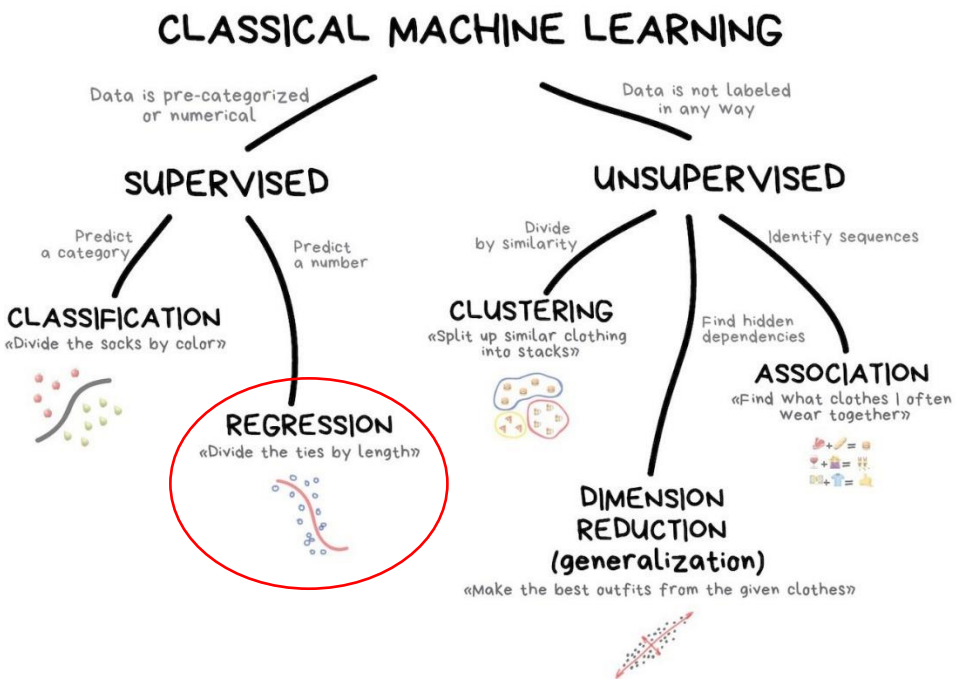
Com relação a banheiros e suítes não foi possível utilizar a mesma técnica, pois ao realizar a subtração de suítes do número de banheiros ocorreram vários registros com número negativo de banheiros. Acreditamos que seja decorrente da falta de critérios padronizados para anúncios, ou seja, enquanto alguns anunciantes incluem os banheiros das suítes no número de banheiros, outros anunciantes informam apenas os banheiros que não estão nas suítes.

Dessa forma, de modo a evitar a distorção dos dados, manteremos os registros de banheiros como estão no original.

5. Criação de Modelos de Machine Learning

Passaremos agora para a fase de modelagem e seguindo o mapa abaixo[21], vemos que se trata de um problema de regressão linear pois os dados são pre-categorizados ou numéricos e almejamos prever um número, qual seja o preço dos apartamentos.

Figura 52 - Mapa Classical Machine Learning



Os algoritmos de regressão linear que vamos utilizar estão disponíveis nas bibliotecas Statsmodel e Scikit-learn e são os seguintes:

1. Ordinary Least Squares (Statsmodel);
2. Ordinary Least Squares (Scikit-learn);
3. Regularized Regression Methods - Ridge Regression (Scikit-learn);
4. Least Absolute Shrinkage And Selection Operator regularization - LASSO (Scikit-learn);
5. Elastic Net (Scikit-learn);
6. Support Vector Regression - SVM-LinearSVR (Scikit-learn); e
7. Stochastic Gradient Descent: Regressor - SGDRegressor (Scikit-learn)

[21] Disponível em https://vas3k.com/blog/machine_learning/?source=post_page-----885aa35db58b----------

Embora contenha o mesmo algoritmo, os itens 1) e 2), além de se referir a bibliotecas diferentes, também tem o diferencial de que o algoritmo no item 2) será utilizado após a divisão dos dados em dados de treinamento e teste. Por esse motivo vamos utilizá-lo duas vezes.

Então vamos a apresentação dos detalhes de cada um deles.

5.1 Ordinary Least Squares (Statsmodel)

O que há de diferente nesse modelo dos modelos preliminares é que agora os dados já estão tratados e com o problema de dupla contagem resolvido.

Vamos inicialmente fazer uma cópia do dataset, a qual chamaremos de “Dados_copy” e, em seguida alterar o tipo de dado da variável “bairro” para “inteiros” por meio do algoritmo “LabelEncoder” da biblioteca Sklearn. A alteração do tipo de variável é devida para que o algoritmo de regressão produza um único coeficiente para essa variável

Em testes anteriores com a variável “bairro” do tipo category tínhamos um coeficiente para cada nome de bairro e, desse modo, ficava difícil excluir os itens com “Valor P” maior do que 0,05. Seguem os códigos:

Figura 53 - Visualização de colunas, criação da cópia e visualização dos tipos de variáveis

```
In [427]: Dados.columns

Out[427]: Index(['preco', 'area', 'quartos', 'banheiros', 'vagas', 'suites', 'bairro',
                 'População', 'População(%)', 'Area_bairro', 'Área(%)',
                 'Densidade_populacional', 'preco_metro', 'preco_ajustado_mod_pre',
                 'preco_so', 'outlier'],
                dtype='object')

In [428]: #Criando uma cópia do dataset
          Dados_copy = Dados

In [430]: Dados_copy.dtypes

Out[430]: preco                      int64
          area                     float64
          quartos                  float64
          banheiros                float64
          vagas                    float64
          suites                   float64
          bairro                  category
          População                  int64
          População(%)             float64
          Area_bairro                int64
          Área(%)                  float64
          Densidade_populacional   float64
          preco_metro              float64
          preco_ajustado_mod_pre   float64
          preco_so                 float64
          outlier                   object
          dtype: object
```

Figura 54 - Transformação dos dados da variável "bairro"

```
In [431]: #Convertendo a coluna bairro para inteiros
          from sklearn.preprocessing import LabelEncoder
          labelencoder = LabelEncoder()
          Dados_copy['bairro'] = labelencoder.fit_transform(Dados_copy['bairro'])
          Dados_copy.dtypes

Out[431]: preco                    int64
          area                   float64
          quartos                float64
          banheiros              float64
          vagas                  float64
          suites                 float64
          bairro                   int32
          População                int64
          População(%)           float64
          Area_bairro              int64
          Área(%)                float64
          Densidade_populacional float64
          preco_metro            float64
          preco_ajustado_mod_pre float64
          preco_so               float64
          outlier                 object
          dtype: object
```

Abaixo a criação e treinamento do modelo 1, utilizando como variável target a coluna “preco_so” que contém os dados de preços de apartamento sem os outliers. Também, temos a invocação do summary com o resultado do treinamento (ajuste).

```
In [432]: #Criação do modelo 1 utilizando a biblioteca statsmodel
          #(apenas com os atributos mais relevantes => Valor P < 0,05)
          import statsmodels.formula.api as sm
          modelo_1 = sm.ols(formula = 'preco_so ~ area + quartos + banheiros + bairro + vagas + suites + População + Area_bairro '
                            '+ Densidade_populacional + preco_metro', data = Dados_copy)
          modelo_1_treinado = modelo_1.fit()
          modelo_1_treinado.summary()
```

Na sequência temos o resultado do treinamento/ajuste do modelo 1.

OLS Regression Results

Dep. Variable:	preco_so	**R-squared:**	0.969
Model:	OLS	**Adj. R-squared:**	0.968
Method:	Least Squares	**F-statistic:**	8068.
Date:	Fri, 23 Apr 2021	**Prob (F-statistic):**	0.00
Time:	09:27:41	**Log-Likelihood:**	-34396.
No. Observations:	2633	**AIC:**	6.881e+04
Df Residuals:	2622	**BIC:**	6.888e+04
Df Model:	10		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-7.603e+05	3.43e+04	-22.169	0.000	-8.28e+05	-6.93e+05
area	5827.5614	53.483	108.960	0.000	5722.688	5932.435
quartos	1.285e+04	4211.111	3.052	0.002	4594.399	2.11e+04
banheiros	-6128.2527	2218.037	-2.763	0.006	-1.05e+04	-1778.972
bairro	6265.7577	1504.917	4.164	0.000	3314.812	9216.703
vagas	-1.189e+04	4610.098	-2.580	0.010	-2.09e+04	-2853.173
suites	6.051e+04	6180.281	9.791	0.000	4.84e+04	7.26e+04
População	6.6960	0.785	8.532	0.000	5.157	8.235
Area_bairro	-34.3585	4.312	-7.969	0.000	-42.813	-25.904
Densidade_populacional	-3.875e+04	4346.623	-8.915	0.000	-4.73e+04	-3.02e+04
preco_metro	157.9197	1.597	98.911	0.000	154.789	161.050

Omnibus:	1209.736	**Durbin-Watson:**	1.951
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	19800.777
Skew:	-1.752	**Prob(JB):**	0.00
Kurtosis:	15.969	**Cond. No.**	6.10e+05

Interessante observar uma pequena melhoria do Coeficiente de Determinação (R^2) de 0.968 para 0.969. Os critérios de informação AIC[22] (Akaike Information Criterion) e BIC (Bayesian Information Criterion[23]) também tiveram uma pequena melhora de 6.884e+04 para 6.881e-04, e, de 6.889e+04 para 6.888e+04, respectivamente.

Embora seja comum utilizar os critérios de informação para comparação de modelos, vamos nos concentrar para efeito de comparação no Coeficiente de Determinação (R^2).

O diferencial entre esses dois modelos foi a inclusão das variáveis “quartos”, após o tratamento da dupla contagem, e a variável “bairro” após a transformação dos dados para o tipo “inteiro”.

5.2 Ordinary Least Squares (Scikit-learn)

Dessa vez vamos utilizar o mesmo algoritmo, só que fornecido pela biblioteca Scikit-learn.

Vamos utilizar o dataset “Dados” e dividi-lo em variáveis preditoras (X_Dados) e variável target (y_Dados). Na construção de X_Dados teremos o cuidado para remover as variáveis que não tem capacidade preditora (P value >0,05), já verificado em etapas anteriores, e igualando as features com o dataset “Dados_copy” usado no modelo 1. Seguem os códigos:

Figura 55 - Divisão do dataset em variáveis preditoras e variável target

```
In [433]: Dados.columns

Out[433]: Index(['preco', 'area', 'quartos', 'banheiros', 'vagas', 'suites', 'bairro',
                 'População', 'População(%)', 'Area_bairro', 'Área(%)',
                 'Densidade_populacional', 'preco_metro', 'preco_ajustado_mod_pre',
                 'preco_so', 'outlier'],
                dtype='object')

In [435]: X_Dados = Dados.drop(columns = ['preco','preco_ajustado_mod_pre', 'outlier', 'preco_so',
                                          'População(%)','Área(%)'], axis = 1) # todas as variáveis/atributos, exceto 'preco_so'
          y_Dados = Dados['preco_so'] # variável-target
```

Também faremos aqui o tratamento da variável “bairro” utilizando a ferramenta LabelEncoder.

[22] Disponível em https://en.wikipedia.org/wiki/Akaike_information_criterion
[23] Disponível em https://en.wikipedia.org/wiki/Bayesian_information_criterion

Figura 56 - Transformação da variável "bairro" para o tipo "inteiro"

```
In [436]: labelencoder = LabelEncoder()
          X_Dados['bairro'] = labelencoder.fit_transform(X_Dados['bairro'])

          X_Dados
```

Out[436]:

	area	quartos	banheiros	vagas	suites	bairro	População	Area_bairro	Densidade_populacional	preco_metro
0	63.00	1.0	2.0	1.0	1.0	6	42038	5838	7.2	4539.682540
1	75.00	2.0	2.0	1.0	1.0	6	42038	5838	7.2	4373.333333
2	62.77	1.0	2.0	2.0	1.0	6	42038	5838	7.2	5894.535606

Na sequência, vamos fazer a divisão dos conjuntos de dados X_Dados e y_Dados em dados de treinamento e dados de testes por meio da ferramenta “train_test_split” da biblioteca Sklearn. Após essa divisão temos 2.106 registros para treinamento e 527 para teste.

Essa divisão se faz necessária para podermos avaliar o modelo com dados por ele desconhecidos. Assim evitamos problemas como “overfitting”, ou seja, o super ajustamento do modelo aos dados de treinamento, acarretando uma redução na capacidade de generalização e previsão.

Figura 57 - Divisão em dados de treinamento e teste

```
In [437]: # Definindo os dataframes de treinamento e teste:
          from sklearn.model_selection import train_test_split

          X_treinamento, X_teste, y_treinamento, y_teste = train_test_split(X_Dados,
                                                                            y_Dados,
                                                                            test_size = 0.2,
                                                                            random_state = 2208)
          print(f"Dataframe de treinamento: {X_treinamento.shape[0]} linhas")
          print(f"Dataframe de teste......: {X_teste.shape[0]} linhas")
```

```
Dataframe de treinamento: 2106 linhas
Dataframe de teste......: 527 linhas
```

Continuando, já podemos importar a biblioteca com o algoritmo “LinearRegression” e instanciar o objeto na variável “modelo_2”. Não vamos definir parâmetros diferentes dos predefinidos (default).

```
In [438]: #Importa a biblioteca e instancia o objeto
          from sklearn.linear_model import LinearRegression
          modelo_2 = LinearRegression()
```

Faremos o ajuste do modelo com os dados de treinamento (X_treinamento e y_treinamento).

```
In [440]: # Treina o modelo usando as amostras/dataset de treinamento: X_treinamento e y_treinamento
          modelo_2.fit(X_treinamento, y_treinamento)
```

```
Out[440]: LinearRegression()
```

Podemos verificar abaixo o valor do intercepto e dos coeficientes obtido pelo modelo 2.

Figura 58 - Valor do Intercepto e coeficientes

```
In [441]: # Valor do intercepto
          modelo_2.intercept_

Out[441]: -740639.1685913978

In [442]: # Coeficientes do modelo de Regressão Linear
          coeficientes_regressao_linear = pd.DataFrame([X_treinamento.columns, modelo_2.coef_]).T
          coeficientes_regressao_linear = coeficientes_regressao_linear.rename(columns={0: 'Feature/variável/coluna', 1: 'Coeficientes'})
          coeficientes_regressao_linear
```

Out[442]:

	Feature/variável/coluna	Coeficientes
0	area	5845.5
1	quartos	10375.8
2	banheiros	-3720.08
3	vagas	-13704.8
4	suites	53349
5	bairro	5905.51
6	População	6.82118
7	Area_bairro	-36.5169
8	Densidade_populacional	-39976.9
9	preco_metro	159.237

E agora, temos os valores do Coeficiente de Determinação (R^2) calculado com os dados de treinamento e, também, com os dados de teste utilizando o modelo 2.

Figura 59 - Coeficiente de Determinação (R^2)

```
In [443]: # Score (R²)
          print('Score com dados de treinamento: %f.\nScore com dados de teste: %f'%
                (modelo_2.score(X_treinamento,y_treinamento),modelo_2.score(X_teste,y_teste)))

          Score com dados de treinamento: 0.969192.
          Score com dados de teste: 0.965579
```

Verifica-se que, com os dados de teste há uma pequena piora do R^2, o que é normal de se esperar. Entretanto, entendemos que essa redução no R^2 é pequena, o que demonstra a alta capacidade de generalização do modelo.

Comparando com o modelo 1, o qual tem o mesmo valor de R^2, vemos que há pouco espaço para melhora, o que tentaremos a seguir por meio de outros algoritmos de regressão linear.

5.3 Regularized Regression Methods - Ridge Regression (Scikit-learn)

O algoritmo Ridge Regression é utilizado para resolver um dos problemas dos algoritmos anteriores (OLS), pois ele impõe uma penalidade no tamanho dos

coeficientes, fazendo com que atributos correlacionados tenham coeficientes parecidos[24].

Vamos primeiro trabalhar em um modelo mais simples, importando o algoritmo Ridge da biblioteca Sklearn e instanciando o objeto. Como parâmetro definiremos alpha igual a 0,1.

```
In [323]: #Importação da biblioteca e criação do modelo e definição do valor de alpha
          from sklearn.linear_model import Ridge
          modelo_3 = Ridge(alpha = 0.1)
```

Treinamos o modelo com o código abaixo.

```
In [324]: #Treinamento do modelo
          modelo_3.fit(X_treinamento, y_treinamento)

Out[324]: Ridge(alpha=0.1)
```

E verificamos os coeficientes.

```
In [595]: # Lista das variáveis + coeficientes da Ridge:
          list(zip(X_treinamento.columns, (modelo_3.coef_)))

Out[595]: [('area', 5845.5476464823205),
           ('quartos', 10365.005068931343),
           ('banheiros', -3717.577295054702),
           ('vagas', -13698.661091137763),
           ('suites', 53328.49200301114),
           ('bairro', 5905.906559658098),
           ('População', 6.820310185993782),
           ('Area_bairro', -36.512587358649256),
           ('Densidade_populacional', -39971.99137569031),
           ('preco_metro', 159.23713249102354)]
```

Por último verificamos o Coeficiente de Determinação (R^2) e percebemos que não houve melhora.

```
In [596]: print('Score com dados de treinamento: %f.\nScore com dados de teste: %f'%
                (modelo_3.score(X_treinamento,y_treinamento),modelo_3.score(X_teste,y_teste)))

          Score com dados de treinamento: 0.969192.
          Score com dados de teste: 0.965579
```

Agora vamos tentar buscar o valor de alpha que maximiza o resultado, utilizando a ferramenta GridSearchCV da biblioteca Sklearn.

Como parâmetros usaremos uma lista de nove valores para alpha que parte de 0,00001 e vai até 1.000, o algoritmo será Ridge, a métrica para score será R^2, e definiremos 10 partições para validação cruzada, conforme abaixo.

[24] Disponível em https://medium.com/turing-talks/turing-talks-20-regress%C3%A3o-de-ridge-e-lasso-a0fc467b5629 e https://scikit-learn.org/stable/modules/linear_model.html

Figura 60 - Busca do melhor alpha

```
In [614]: #Utilizando GridSearchCV para busca do melhor alpha
          from sklearn.model_selection import GridSearchCV
          parameters = {'alpha':[.00001,.0001,.001,.01,.1,1,10,100,1000]}
          modelo_3_1 = Ridge()
          modelo_3_1 = GridSearchCV(modelo_3_1, parameters, scoring='r2', cv=10)
          modelo_3_1.fit(X_treinamento,y_treinamento)

Out[614]: GridSearchCV(cv=10, estimator=Ridge(),
                       param_grid={'alpha': [1e-05, 0.0001, 0.001, 0.01, 0.1, 1, 10, 100,
                                             1000]},
                       scoring='r2')
```

Como resultado do modelo 3.1 temos como o melhor alpha sendo igual a 10 e o melhor score obtido na validação cruzada igual a 0,967728.

```
In [615]: modelo_3_1.best_params_

Out[615]: {'alpha': 10}
```

```
In [616]: modelo_3_1.best_score_

Out[616]: 0.96772802546756
```

Ao avaliar o modelo com dados de treinamento e teste temos os seguintes Coeficientes de Determinação (R^2).

```
In [617]: print('Score com dados de treinamento: %f.\nScore com dados de teste: %f'%
                (modelo_3_1.score(X_treinamento,y_treinamento),modelo_3_1.score(X_teste,y_teste)))

          Score com dados de treinamento: 0.969190.
          Score com dados de teste: 0.965540
```

Percebe-se que houve uma pequena redução do score com dados de treinamento, provavelmente decorrente da aplicação da penalização.

5.4 Least Absolute Shrinkage And Selection Operator regularization - LASSO (Scikit-learn)

Nesse tópico vamos abordar outro algoritmo que busca penalizar os coeficientes com o objetivo de diminuir a variância do modelo. Quando há múltiplas features altamente correlacionadas (ou seja, features que se comportam da mesma maneira) a regularização Lasso seleciona apenas uma dessas features e zera os coeficientes das outras, de forma a minimizar a penalização L1[25].

[25] Disponível em https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html?highlight=lasso#sklearn.linear_model.Lasso

Da mesma forma que no tópico anterior, vamos inicialmente gerar um modelo maios simples. Importaremos o algoritmo LASSO da biblioteca Sklearn, instanciaremos o objeto na variável “modelo_4” e definiremos como parâmetro o valor de alpha igual a 0,1, conforme abaixo.

```
In [618]: #Importação da biblioteca, criação do modelo e definição do valor de alpha
          from sklearn.linear_model import Lasso
          modelo_4 = Lasso(alpha = .1)
```

Em seguida vamos ajustar o modelo aos dados de treinamento e verificar os coeficientes.

```
In [619]: #Treinamento do modelo
          modelo_4.fit(X_treinamento, y_treinamento)

Out[619]: Lasso(alpha=0.1)

In [620]: # Lista das variáveis + coeficientes do Lasso:
          list(zip(X_treinamento.columns, (modelo_4.coef_)))

Out[620]: [('area', 5845.496519602181),
           ('quartos', 10375.064912830696),
           ('banheiros', -3719.8808892022826),
           ('vagas', -13704.256937925184),
           ('suites', 53347.70457022557),
           ('bairro', 5905.492226675107),
           ('População', 6.821134879185218),
           ('Area_bairro', -36.51670670540579),
           ('Densidade_populacional', -39976.65271883965),
           ('preco_metro', 159.23731757457307)]
```

Próximo passo é verificar os scores e mais uma vez não vemos melhoria dos Coeficientes de Determinação (R^2).

```
In [621]: print('Score com dados de treinamento: %f.\nScore com dados de teste: %f'%
                (modelo_4.score(X_treinamento,y_treinamento),modelo_4.score(X_teste,y_teste)))

          Score com dados de treinamento: 0.969192.
          Score com dados de teste: 0.965579
```

Novamente, tentaremos obter o melhor valor de alpha utilizando a ferramenta GridSearchCV. Definiremos os mesmos parâmetros do tópico anterior com exceção do *estimator* que será LASSO, quais sejam: uma lista de nove valores para alpha que parte de 0,00001 e vai até 1.000, a métrica para score será R^2, e 10 partições para validação cruzada. Efetuamos o ajuste do modelo conforme abaixo.

```
In [622]: #Utilizando GridSearch e Cross-Validation para busca do melhor alpha
          parameters = {'alpha':[.00001,.0001,.001,.01,.1,1,10,100,1000]}
          modelo_4_1 = Lasso()
          modelo_4_1 = GridSearchCV(modelo_4_1, parameters, scoring='r2', cv=10)
          modelo_4_1.fit(X_treinamento,y_treinamento)

Out[622]: GridSearchCV(cv=10, estimator=Lasso(),
                       param_grid={'alpha': [1e-05, 0.0001, 0.001, 0.01, 0.1, 1, 10, 100,
                                             1000]},
                       scoring='r2')
```

Com resultado temos o melhor parâmetro para alpha igual a 100 e melhor score igual a 0,967722.

```
In [623]: print('Melhores parâmetros:')
          modelo_4_1.best_params_

          Melhores parâmetros:

Out[623]: {'alpha': 100}
```

```
In [624]: print('Melhor score:')
          modelo_4_1.best_score_

          Melhor score:

Out[624]: 0.9677224672360367
```

Após fazer a validação nos dados de testes temos o seguinte:

```
In [625]: print('Score com dados de treinamento: %f.\nScore com dados de teste: %f'%
                (modelo_4_1.score(X_treinamento,y_treinamento),modelo_4_1.score(X_teste,y_teste)))

          Score com dados de treinamento: 0.969191.
          Score com dados de teste: 0.965556
```

Mais uma vez verificamos que há uma pequena redução no R^2 com dados de treinamento e testes em consequência da penalização imposta pelo algoritmo.

5.5 Elastic Net (Scikit-learn)

Agora vamos utilizar um algoritmo que combina os métodos de regularização apresentados nos últimos dois tópicos[26]. Também iniciaremos com um modelo mais simples e posteriormente tentaremos maximizar o R^2.

Primeiro importaremos o algoritmo ElasticNet da biblioteca Sklearn, instanciaremos o objeto na variável “modelo_5” e definiremos os parâmetros de alpha igual a 0,1 e l1_ratio igual a 1. Na sequência faremos o ajuste do modelo, conforme códigos abaixo.

```
In [630]: #Importa a biblioteca e instancia o objeto
          from sklearn.linear_model import ElasticNet
          modelo_5 = ElasticNet(alpha=.1, l1_ratio=1)
```

```
In [631]: #Treinamento do modelo
          modelo_5.fit(X_treinamento, y_treinamento)

Out[631]: ElasticNet(alpha=0.1, l1_ratio=1)
```

Abaixo podemos conferir os valores dos coeficientes.

[26] Disponível em https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.ElasticNet.html#sklearn.linear_model.ElasticNet

Figura 61 - Coeficientes do modelo_5

```
In [632]: list(zip(X_treinamento,modelo_5.coef_))

Out[632]: [('area', 5845.496519602181),
           ('quartos', 10375.064912830696),
           ('banheiros', -3719.8808892022826),
           ('vagas', -13704.256937925184),
           ('suites', 53347.70457022557),
           ('bairro', 5905.492226675107),
           ('População', 6.821134879185218),
           ('Area_bairro', -36.51670670540579),
           ('Densidade_populacional', -39976.65271883965),
           ('preco_metro', 159.23731757457307)]
```

E por fim verificamos os scores (R^2) com dados de treinamento e teste.

```
In [633]: print('Score com dados de treinamento: %f.\nScore com dados de teste: %f'%
                (modelo_5.score(X_treinamento,y_treinamento),modelo_5.score(X_teste,y_teste)))

          Score com dados de treinamento: 0.969192.
          Score com dados de teste: 0.965579
```

Aparentemente chegamos no limite, pois não conseguimos melhorar os scores. Vamos tentar novamente buscar os melhores parâmetro para o modelo utilizando as ferramentas GridSearchCV. Como parâmetros definiremos uma lista de nove valores para alpha e l1_ratio que parte de 0,00001 e vai até 1.000, a métrica para score será R^2, 10 partições para validação cruzada, número de Jobs igual a -1, significando que serão utilizados todos os processadores, *refit* igual a *True* significando que será reajustado o *estimator* usando os melhores do *dataset* inteiro. O *estimator*, obviamente será o ElasticNet. Vejamos o código.

```
In [646]: #Utilizando GridSearch e Cross-Validation para busca dos melhores parâmetros
          en = ElasticNet()

          # Otimização dos hiperparâmetros:
          d_hiperparametros = {'alpha': [.00001,.0001,.001,.01,.1,1,10,100,1000],
                               'l1_ratio': [.00001,.0001,.001,.01,.1,1,10,100,1000]}

          modelo_5_1 = GridSearchCV(estimator = en, # Elastic Net
                                    param_grid = d_hiperparametros, # Dicionário com os hiperparâmetros
                                    scoring = 'r2',
                                    n_jobs = -1, # Usar todos os processadores/computação
                                    refit = True,
                                    cv = 10) # Número de Cross-Valitations
```

Treinaremos o modelo e conferimos os melhores parâmetros e scores.

Figura 62 - Treinamento do modelo 5.1

```
In [647]: modelo_5_1.fit(X_treinamento,y_treinamento)

Out[647]: GridSearchCV(cv=10, estimator=ElasticNet(), n_jobs=-1,
                       param_grid={'alpha': [1e-05, 0.0001, 0.001, 0.01, 0.1, 1, 10, 100,
                                             1000],
                                   'l1_ratio': [1e-05, 0.0001, 0.001, 0.01, 0.1, 1, 10,
                                                100, 1000]},
                       scoring='r2')

In [648]: modelo_5_1.best_params_

Out[648]: {'alpha': 0.01, 'l1_ratio': 1e-05}

In [649]: modelo_5_1.best_score_

Out[649]: 0.9677309593760455
```

Agora vamos verificar os scores (R^2) nos dados de treinamento e de teste.

```
In [650]: print('Score com dados de treinamento: %f.\nScore com dados de teste: %f'%
              (modelo_5_1.score(X_treinamento,y_treinamento),modelo_5_1.score(X_teste,y_teste)))

          Score com dados de treinamento: 0.969185.
          Score com dados de teste: 0.965497
```

Mais uma vez vemos uma pequena redução nos scores em função das penalizações impostas pelo algoritmo.

5.6 Linear Support Vector Regression - SVM-LinearSVR (Scikit-learn)

Dessa vez utilizaremos o algoritmo Linear Support Vector Regression, o qual se trata de uma versão especializada do algoritmo Support Vector Machine para problemas de regressão linear, pois utiliza apenas o kernel linear. Assim, o algoritmo se torna mais leve e eficiente para escalar mais dados[27].

Seguindo a ordem natural vamos importar a biblioteca e instanciar o objeto. Como parâmetros definiremos, nesse primeiro modelo, C=1 (força do parâmetro de regularização), tol=1e-2 (tolerância para critério de stop), e épsilon=0.1(função de perda).

Figura 63 - Criação a ajuste do modelo 6 - SVR (sklearn)

```
In [696]: #Importa a biblioteca e instancia o objeto
          from sklearn.svm import LinearSVR
          modelo_6 = LinearSVR(random_state=0, C=1, tol=1e-2, epsilon=0.1)

In [697]: #Treinamento do modelo
          modelo_6.fit(X_treinamento, y_treinamento)

Out[697]: LinearSVR(C=1, epsilon=0.1, random_state=0, tol=0.01)
```

[27] Disponível em https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVR.html#sklearn.svm.LinearSVR

Como resultado temos um score muito aquém do obtido nos modelos anteriores.

```
In [698]: print('Score com dados de treinamento: %f.\nScore com dados de teste: %f'%
              (modelo_6.score(X_treinamento,y_treinamento),modelo_6.score(X_teste,y_teste)))

          Score com dados de treinamento: 0.915462.
          Score com dados de teste: 0.908700
```

Na sequência, tentaremos obter os parâmetros ideais, utilizando a ferramenta GridSearchCV. Para seleção dos parâmetros “C”, “tol” e “épsilon”, apresentamos uma lista de múltiplos de 10 que parte de 0,00001 e vai até 1.000. Definimos como critério de “scoring” o R^2, utilizar todos os processadores como o “n_jobs” igual a -1, “refit” igual a “True” para utilizar os melhores parâmetros no dataset inteiro e, também, definimos 10 partições para o cross validation.

Figura 64 - Criação do modelo 6.2 com GridSearchCV

```
In [699]: #Utilizando GridSearch e Cross-Validation para busca dos melhores parâmetros
          svr = LinearSVR()

          # Otimização dos hiperparâmetros:
          d_hiperparametros = {'C': [.00001,.0001,.001,.01,.1,1,10,100,1000],
                               'tol': [.00001,.0001,.001,.01,.1,1,10,100,1000],
                              'epsilon': [.00001,.0001,.001,.01,.1,1,10,100,1000]}

          modelo_6_1 = GridSearchCV(estimator = svr, # Suppot Vector Regressor
                                    param_grid = d_hiperparametros, # Dicionário com os hiperparâmetros
                                    scoring = 'r2',
                                    n_jobs = -1, # Usar todos os processadores/computação
                                    refit = True,
                                    cv = 10) # Número de Cross-Valitations

In [700]: modelo_6_1.fit(X_treinamento,y_treinamento)

Out[700]: GridSearchCV(cv=10, estimator=LinearSVR(), n_jobs=-1,
                       param_grid={'C': [1e-05, 0.0001, 0.001, 0.01, 0.1, 1, 10, 100,
                                         1000],
                                   'epsilon': [1e-05, 0.0001, 0.001, 0.01, 0.1, 1, 10,
                                               100, 1000],
                                   'tol': [1e-05, 0.0001, 0.001, 0.01, 0.1, 1, 10, 100,
                                           1000]},
                       scoring='r2')
```

Após o treinamento obtivemos como melhores parâmetros “C” igual a 1, épsilon igual a 1000 e tolerância igual a 0.0001, sendo que não houve uma melhora no score, mas, pelo contrário, houve uma piora significativa.

```
In [701]: modelo_6_1.best_params_

Out[701]: {'C': 1, 'epsilon': 1000, 'tol': 0.0001}

In [702]: modelo_6_1.best_score_

Out[702]: 0.9149438796582515

In [703]: print('Score com dados de treinamento: %f.\nScore com dados de teste: %f'%
              (modelo_6_1.score(X_treinamento,y_treinamento),modelo_6_1.score(X_teste,y_teste)))

          Score com dados de treinamento: 0.865599.
          Score com dados de teste: 0.855244
```

Deixando esse modelo de lado, partiremos para o próximo.

5.7 Stochastic Gradient Descent: Regressor - SGDRegressor (Scikit-learn)

Esse modelo linear, durante a descida do gradiente, calcula a função de perda (*loss function*) de forma estocástica, ou seja, a cada exemplo[28].

Primeiro vamos fazer um modelo inicial usando parâmetros arbitrários. Nesse modelo vamos utilizar outras duas ferramentas disponível na biblioteca sklearn, a "make_pipeline" e "StandardScaler".

A ferramenta "make_pipeline"[29] tem por objetivo organizar um pipeline, ou seja, uma ordem de passos ou comandos a ser seguido pelo script. No caso queremos primeiro escalar os dados de forma padronizada, ou seja, transformá-los na escala padrão, para depois fazer o ajuste do modelo.

E é essa transformação dos dados que é realizada pela ferramenta "StandardScaler"[30]. Em suma, a ferramenta realizará o seguinte cálculo nos dados das variáveis preditoras:

z = (x - u) / s

onde:

x é a variável a ser transformada;

u é a média dos exemplos (x);

s é o desvio padrão dos exemplos (x)

Os parâmetros definidos para o modelo foram "penalty" igual a "elasticnet", "max_iter" igual a 1000 e "tol" igual a 0,001, conforme abaixo:

[28] Disponível em https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.html#sklearn.linear_model.SGDRegressor

[29] Disponível me https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.make_pipeline.html?highlight=make_pipeline#sklearn.pipeline.make_pipeline

[30] Disponível em https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html?highlight=standardscaler#sklearn.preprocessing.StandardScaler

Figura 65 - Criação do modelo 7 com o uso das ferramentas make_pipeline, StandardScaler e SCDRegressor

```
In [788]: #importando as bibliotecas
          from sklearn.linear_model import SGDRegressor
          from sklearn.pipeline import make_pipeline
          from sklearn.preprocessing import StandardScaler

          # intanciando o objeto com a criação de pipeline, padronização das variáveis preditoras e treinamento do modelo
          modelo_7 = make_pipeline(StandardScaler(),SGDRegressor(penalty='elasticnet', max_iter=1000, tol=1e-3))
          modelo_7.fit(X_treinamento, y_treinamento)

Out[788]: Pipeline(steps=[('standardscaler', StandardScaler()),
                          ('sgdregressor', SGDRegressor(penalty='elasticnet'))])
```

O resultado (R^2) ficou uma pouco abaixo dos melhores modelos até agora.

```
In [789]: print('Score com dados de treinamento: %f.\nScore com dados de teste: %f'%
                (modelo_7.score(X_treinamento,y_treinamento),modelo_7.score(X_teste,y_teste)))

          Score com dados de treinamento: 0.968911.
          Score com dados de teste: 0.965616
```

Na sequência, vamos tentar obter os melhores parâmetros com o uso da ferramenta GridSearchCV. Definiremos os seguintes parâmetros no GridSearchCV: *estimator* igual a SGDRegressor; *scoring* igual a “r2”; n_*jobs* igual a -1, indicando o uso de todos os processadores; cv igual a 10, ou seja, 10 partições para cross validation; e, no grid, como parâmetros do estimator, temos para “penalty” as *strings* “l2”, “l1” e “elasticnet”; para valores de “alpha” uma lista que vai de 0,00001 até 1.000; e para valores de “l1_ratio” uma lista que vai de 0,00001 até 1. Segue o código.

Figura 66 - Criação do modelo 7.1 com o uso de GridSearchCV e SGDRegressor

```
In [790]: #Utilizando GridSearch e Cross-Validation para busca dos melhores parâmetros
          sgd = SGDRegressor()

          # Otimização dos hiperparâmetros:
          d_hiperparametros = {'penalty': ['l2','l1','elasticnet'],
                               'alpha': [.00001,.0001,.001,.01,.1,1,10,100,1000],
                               'l1_ratio':[.00001,.0001,.001,.01,.1,1]}

          modelo_7_1 = GridSearchCV(estimator = sgd, # SGDRegressor
                                    param_grid = d_hiperparametros, # Dicionário com os hiperparâmetros
                                    scoring = 'r2',
                                    n_jobs = -1, # Usar todos os processadores/computação
                                    cv = 10) # Número de Cross-Valitations
```

Como melhores parâmetros e melhor score temos o seguinte:

```
In [792]: modelo_7_1.best_params_

Out[792]: {'alpha': 1e-05, 'l1_ratio': 0.01, 'penalty': 'elasticnet'}

In [793]: modelo_7_1.best_score_

Out[793]: 0.9677123296262943
```

Agora vamos padronizar as variáveis preditoras e criar outro modelo apenas com os melhores parâmetros e visualizar o score. Seguem os códigos.

Figura 67 - Criação do modelo 7.2 com melhores parâmetros

```
In [794]: #Padronização das variáveis preditoras de treinamento e teste por necessidade do algoritmo
          SS = StandardScaler()
          SSTrein = SS.fit_transform(X_treinamento, y=y_treinamento)
          SSTest = SS.fit_transform(X_teste, y=y_teste)

In [795]: #Instancia o objeto (criação do modelo com os melhores parâmetros)
          modelo_7_2 = SGDRegressor(alpha= 0.1, l1_ratio= 0.0001, penalty= 'l1')

In [796]: #Ajuste do modelo
          modelo_7_2.fit(SSTrein, y_treinamento)

Out[796]: SGDRegressor(alpha=0.1, l1_ratio=0.0001, penalty='l1')

In [797]: print('Score com dados de treinamento: %f.\nScore com dados de teste: %f'%
                (modelo_7_2.score(SSTrein,y_treinamento),modelo_7_2.score(SSTest,y_teste)))

          Score com dados de treinamento: 0.968972.
          Score com dados de teste: 0.961960
```

Vemos acima que o score (R^2) ficou próximo aos modelos já treinados em tópicos anteriores, o que pode indicar que chegamos no limite de melhoria.

6. Apresentação dos Resultados

Nesta seção vamos apresentar os resultados obtidos por meio do modelo de Canvas proposto por Vasandani, por um comparativo entre os modelos e, também, por meio de um *dashboard* elaborado por meio da biblioteca Bokeh[31].

6.1 Workflow da análise de dados

Segue abaixo um Canvas onde podemos visualizar o *workflow* do trabalho desenvolvido neste TCC.

1. Definição do Problema (*Problem Statement*): Analisar os dados de preços de imóveis (apartamentos) de sete bairros da cidade de Curitiba-PR, bem como dados referentes ao perfil demográfico (densidade populacional) para fins de precificação dos imóveis.	**2. Resultados e previsões (*Outcomes/Predictions*):** Estima-se que os atributos inerentes aos imóveis, como área, número de quartos etc., sejam preponderantes no seu preço. Buscaremos investigar o quanto dados geográficos referente à localização dos apartamentos como densidade populacional podem afetar o preço.	**3. Aquisição dos dados (*Data Acquisition*):** Os dados foram obtidos de duas fontes distintas. Uma com os dados de anúncios de imóveis disponível em uma disciplina de estatística da UFPG e outra com dados geográficos oriundos do Censo de 2010 inseridas em um estudo do DIEESE de 2016. Apesar de serem de períodos diversos, o resultado da análise não será afetado.
4. Modelagem (*Modeling*): Após o devido tratamento dos dados, utilizando -se de abordagens diferenciadas para tratamento de *outliers*, foi realizada a modelagem com diversos algoritmos de regressão linear disponíveis nas bibliotecas Statsmodel e Scikit-Learn.	**5. Avaliação do modelo (*Model Evaluation*):** Os modelos foram avaliados utilizando como *score* o Coeficiente de Determinação (R^2), sendo que em geral os resultados foram similares, indicando um grande ajustamento do modelo aos dados, tanto de treinamento quanto de teste.	**6. Preparação dos dados (*Data Preparation*):** Os dados ausentes, duplicados e outliers foram devidamente tratados de acordo com a técnica pertinente. Por diversas vezes foi preferível excluir os registros para evitar a distorção dos dados.

[31] Disponível em https://docs.bokeh.org/en/latest/index.html

6.2 Comparativo dos modelos

Feita a apresentação inicial do workflow, vamos agora comparar os modelos, inicialmente verificando seus coeficientes na tabela abaixo que foi confeccionada pelo seguinte código:

```
In [816]: #Comparativo dos Coeficientes
          Coeficientes = pd.DataFrame(data=list(zip(X_treinamento,modelo_1_treinado.params[1:],modelo_2.coef_,
                                                    modelo_3.coef_,modelo_4.coef_,modelo_5.coef_,modelo_6.coef_,modelo_7_2.coef_),),)
          Coeficientes.columns=['Atributo','OLS - Statsmodel','OLS - Sklearn', 'Ridge','LASSO','ElasticNet','SVR','SGDRegressor']
          Coeficientes
```

Figura 68 - Coeficientes dos modelos

	Atributo	OLS - Stastmodel	OLS - Sklearn	Ridge	LASSO	ElasticNet	SVR	SGDRegressor
0	area	5827.561381	5845.495818	5845.547646	5845.496520	5845.496520	5891.327686	477362.702625
1	quartos	12851.837396	10375.848203	10365.005069	10375.064913	10375.064913	-127.404998	11378.486889
2	banheiros	-6128.252698	-3720.083338	-3717.577295	-3719.880889	-3719.880889	43.100127	-5064.527169
3	vagas	-11892.973319	-13704.846513	-13698.661091	-13704.256938	-13704.256938	50.141509	-10895.038402
4	suites	60513.987366	53348.950632	53328.492003	53347.704570	53347.704570	73.484069	51204.986383
5	bairro	6265.757746	5905.511657	5905.906560	5905.492227	5905.492227	-44.221628	14496.227560
6	População	6.696014	6.821183	6.820310	6.821135	6.821135	-8.744598	35855.371174
7	Area_bairro	-34.358526	-36.516945	-36.512587	-36.516707	-36.516707	-45.795687	-34223.517446
8	Densidade_populacional	-38751.127201	-39976.906364	-39971.991376	-39976.652719	-39976.652719	-210.108435	-60894.685671
9	preco_metro	157.919725	159.237398	159.237132	159.237318	159.237318	117.489100	277904.665250

Podemos ver nos coeficientes que os modelos OLS – Sklearn, Ridge, LASSO e ElasticNet possuem pequenas divergências entre eles devidos aos parâmetros de regularização. Entretanto, podemos dizer que os coeficientes desses modelos possuem praticamente os mesmos valores.

Os coeficientes do modelo OLS – Statsmodel também estão próximos dos citados no parágrafo anterior, as divergências podem ser atribuídas ao procedimento de separação dos dados em dados de treinamento e dados de teste.

Já o modelo SVR apresentou o coeficiente de determinação (R^2) muito abaixo dos demais, provavelmente por isso que seus coeficientes estão tão diferentes dos outros modelos.

E, por último, quanto ao modelo SGDRegressor podemos afirmar que os coeficientes estão considerando os dados padronizados, com a média em zero, sendo por isso que os coeficientes possuem valor tão elevado. Assim, não servem para efeito de comparação com os demais modelos.

Agora veremos como ficou o *score* dos modelos considerando o coeficiente de determinação (R^2). Cabe registrar que foram selecionados os melhores *scores* de cada algoritmo nos dados de treinamento, independentemente de ser um modelo inicial ou um modelo mais bem trabalhado.

A tabela abaixo apresenta os *scores* por modelo:

Figura 69 - Scores dos modelos (R^2).

	Modelo	Dados de Treinamento	Dados de Teste
0	OLS - Statsmodel	0.968525	0.000000
1	OLS - Sklearn	0.969192	0.965579
2	Ridge	0.969192	0.965579
3	LASSO	0.969192	0.965579
4	ElasticNet	0.969192	0.965579
5	SVR	0.915462	0.908700
6	SGDRegressor	0.968911	0.965616

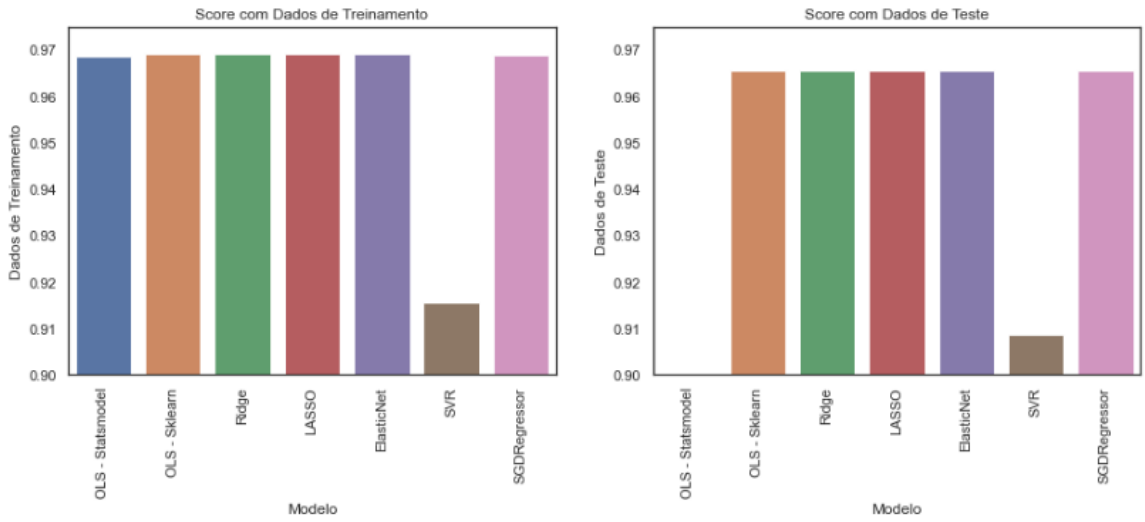
O modelo OLS – Statsmodel foi treinado com toda a base de dados, ou seja, não foi dividida a base em treinamento e teste. Por esse motivo que não há *score* para dados de teste.

Podemos ver que os modelos OLS – Sklearn, Ridge, LASSO e Elasticnet apresentaram o melhor desempenho com dados de treinamento. Já com os dados de teste o melhor modelo foi o SGDRegressor, embora tenha um resultado ligeiramente inferior aos anteriores nos dados de treinamento.

O pior modelo no quesito *score* foi o SVR, tanto nos dados de treinamento, quanto nos dados de teste. Estima-se que tal resultado inferior seja devido a uma maior dificuldade de escolha dos parâmetros ideais e não ao algoritmo em si.

O gráfico a seguir apresenta o mesmo resultado da tabela acima para fins de melhor visualização. Percebe-se que os *scores* estão, aparentemente, no limite de melhoria e que avanços adicionais talvez tenham que ser obtidos por meio de *features engineering*, ou seja, já na fase de coleta dos dados.

Figura 70 - Scores dos modelos (R^2)



Abaixo podemos ver os códigos para a construção da tabela e do gráfico com os *scores.*

```
In [831]: #Comparativo dos Scores
Scores = pd.DataFrame(data=[('OLS - Statsmodel',modelo_1_treinado.rsquared,0),
                            ('OLS - Sklearn',modelo_2.score(X_treinamento,y_treinamento),modelo_2.score(X_teste,y_teste)),
                            ('Ridge',modelo_3.score(X_treinamento,y_treinamento),modelo_3.score(X_teste,y_teste)),
                            ('LASSO',modelo_4.score(X_treinamento,y_treinamento),modelo_4.score(X_teste,y_teste)),
                            ('ElasticNet',modelo_5.score(X_treinamento,y_treinamento),modelo_5.score(X_teste,y_teste)),
                            ('SVR',modelo_6.score(X_treinamento,y_treinamento),modelo_6.score(X_teste,y_teste)),
                            ('SGDRegressor',modelo_7.score(X_treinamento,y_treinamento),modelo_7.score(X_teste,y_teste)),
                            ],columns=['Modelo','Dados de Treinamento', 'Dados de Teste'])
Scores
```

```
In [832]: #Construção do gráfico com scores
plt.figure(figsize=(15.0,5.0), )
plt.subplot(1,2,1)
sns.barplot(x=Scores['Modelo'],y=Scores['Dados de Treinamento'],)
plt.xticks(rotation='vertical')
plt.ylim(bottom=.90, top=.975)
plt.title('Score com Dados de Treinamento')
plt.subplot(1,2,2)
sns.barplot(x=Scores['Modelo'],y=Scores['Dados de Teste'])
plt.xticks(rotation='vertical')
plt.ylim(bottom=.90, top=.975)
plt.title('Score com Dados de Teste')
```

6.3 *Dashboard*

Por último vamos visualizar um *dashboard* desenvolvido por meio da biblioteca Bokeh[32], o qual se trata de um arquivo no formato .html.

Inicialmente, para a elaboração dos gráficos, foram efetuadas algumas alterações no *dataframe* para melhor visualização, como mudança no nome dos bairros. Os códigos podem ser vistos abaixo:

[32] Disponível em https://docs.bokeh.org/en/latest/index.html

Figura 71 - Alterações no dataframe para melhor visualização

```
In [305]: Dados_viz['bairro'].unique()

Out[305]: ['portao', 'agua-verde', 'centro', 'ecoville', 'batel', 'cabral', 'cristo-rei']
          Categories (7, object): ['portao', 'agua-verde', 'centro', 'ecoville', 'batel', 'cabral', 'cristo-rei']

In [346]: #Substituição dos rótulos de bairros para apresentação nos gráficos
          Dados_viz['bairro']=Dados_viz['bairro'].replace('agua-verde','Água-Verde'
                                                  ).replace('batel','Batel'
                                                          ).replace('cristo-rei','Cristo-Rei'
                                                                  ).replace('ecoville','Ecoville'
                                                                          ).replace('portao','Portão'
                                                                                  ).replace('centro','Centro'
                                                                                          ).replace('cabral','Cabral')
```

A seguir foram realizadas as importações da biblioteca Bokeh.

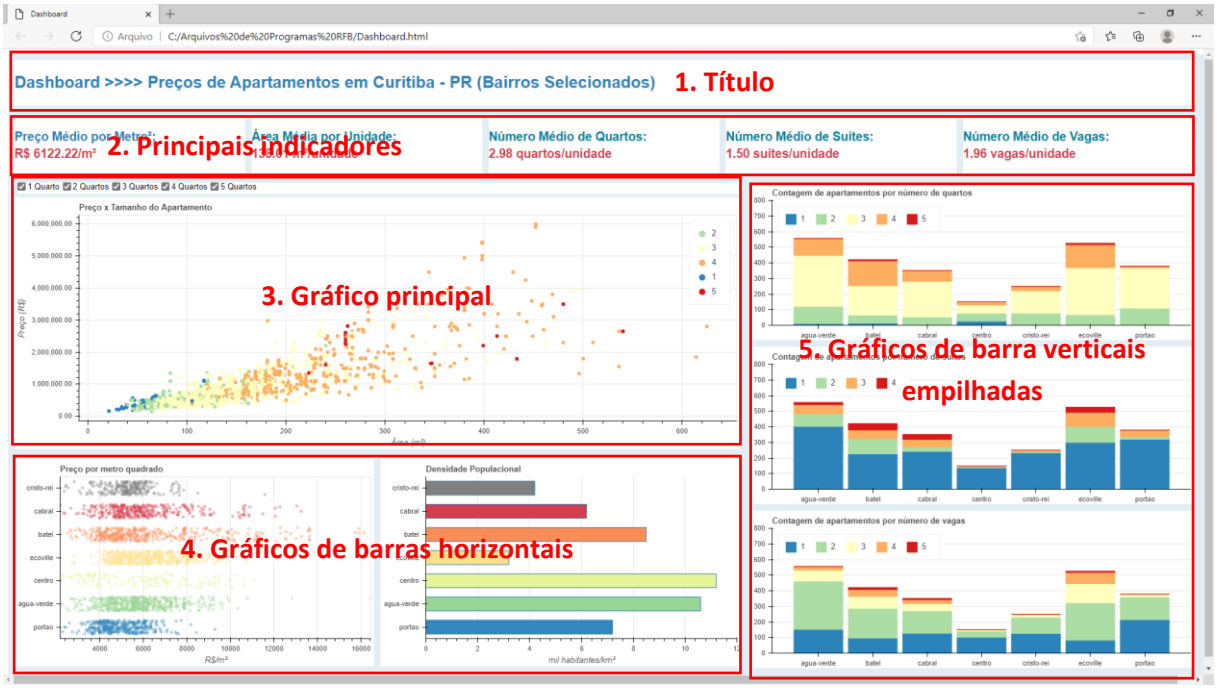
Figura 72 - Importações de bibliotecas

```
: #Importação das bibliotecas
  from bokeh.io import curdoc
  from bokeh.layouts import column, row, WidgetBox
  from bokeh.models import ColumnDataSource, Slider, TextInput, CheckboxGroup, Div
  from bokeh.plotting import figure, output_file, show
  from bokeh.models import CustomJS, ColumnDataSource, CheckboxGroup, Column, Line, Circle, Row, NumeralTickFormatter
  from bokeh.plotting import figure, show
  from bokeh.transform import jitter
  from bokeh.transform import factor_cmap
  from bokeh.palettes import Category10_10, Spectral6, Spectral5,Spectral8, Spectral4
  from bokeh.models.ranges import Range1d
```

Este *dashboard* possui cinco seções conforme abaixo:

1. Título (no topo);
2. Principais indicadores (logo abaixo do título);
3. Gráfico principal interativo, com dados de preço x área (centro);
4. Gráficos de barras horizontais (2), um com dados de preço/m² e outro com densidade populacional segregados por bairro (à esquerda e abaixo); e
5. Gráficos de barras verticais empilhadas (3), com dados de quartos, suítes e vagas de estacionamentos, segregados por bairros (abaixo e à direita).

Figura 73 – Dashboard – mapa das seções



A seguir teceremos comentários sobre cada seção.

O título está localizado em uma “Div” na parte superior do *dashboard* e foi construído com o código abaixo. Parte deste código é composto por linguagem Python e parte por HTML na *string* passada no parâmetro *text da Div.*

Figura 74 - Código para geração do título

```
heading = Div(text="""<h1><b> <font color="#2b83ba">   Dashboard >>>>   Preços de Apartamentos em Curitiba - PR</b>
            (Bairros Selecionados)</font></h1>""",align='center', css_classes=['myclass'],
             background='white', width_policy='max', min_width=1870, max_width=1870, width=1870)
```

Detalhe do título.

Figura 75 - Título

Os principais indicadores são em número de cinco e estão apresentados em outra *Div* logo abaixo do título. Contém informações gerais sobre a média dos principais componentes das unidades (apartamentos), considerando todos os registros do *dataset.* Da mesma forma que no título, contém fragmentos de ambas as linguagens de programação (python e html). Segue o código:

Figura 76 - Código para geração dos principais indicadores

```
largura_kpi = 366

alfa = (Dados_viz['preco'].sum()/Dados_viz['area'].sum()).round(2)
kpi = Div(text="""<h2 align="center"><b><font color="#2b83ba">   Preço Médio por Metro²:</font></b><br><font color="#d53e4f">
          R$ %s/m²</font></h2>"""%alfa, align='center', width_policy='max', min_width=largura_kpi, max_width=largura_kpi,
          width=largura_kpi, sizing_mode="scale_height", css_classes=['myclass'],background='white')

beta = Dados_viz['area'].mean()
kpi1 = Div(text="""<h2><b><font color="##2b83ba">   Área Média por Unidade:</font></b><br><font color="#d53e4f">
           %.2f m²/unidade </font></h2>"""%round(beta,2), align='center', width_policy='max', min_width=largura_kpi,
           max_width=largura_kpi, width=largura_kpi, sizing_mode="scale_height", css_classes=['myclass'],background='white')

charlie= Dados_viz['quartos'].mean()
kpi2 = Div(text="""<h2><b><font color="##2b83ba"> Número Médio de Quartos:</font></b><br><font color="#d53e4f">
           %.2f quartos/unidade </font></h2>"""%round(charlie,2), align='center', width_policy='max',
           min_width=largura_kpi, max_width=largura_kpi, width=largura_kpi, sizing_mode="scale_height",
           css_classes=['myclass'],background='white')

delta= Dados_viz['suites'].mean()
kpi3 = Div(text="""<h2><b><font color="##2b83ba"> Número Médio de Suítes:</font></b><br><font color="#d53e4f">
            %.2f suítes/unidade </font></h2>"""%round(delta,2), align='center', width_policy='max',
           min_width=largura_kpi, max_width=largura_kpi, width=largura_kpi, sizing_mode="scale_height",
           css_classes=['myclass'],background='white')

echo= Dados_viz['vagas'].mean()
kpi4 = Div(text="""<h2><b><font color="##2b83ba"> Número Médio de Vagas:</font></b><br><font color="#d53e4f">
           %.2f vagas/unidade </font></h2>"""%round(echo,2), align='center', width_policy='max',
           min_width=largura_kpi, max_width=largura_kpi, width=largura_kpi, sizing_mode="scale_height",
           css_classes=['myclass'],background='white')
```

Detalhe dos principais indicadores.

Figura 77 - Principais indicadores

O gráfico principal conta com um CheckboxGroup onde é permitido aos usuários selecionarem a quantidade de quartos que desejam visualizar no gráfico. Também conta com uma *hover* com informações numéricas sobre área e preço. As cores são definidas usando um *hue* com a paleta *Spectral5*, facilitando a distinção da quantidade de quartos por apartamento e harmonizando com o restante do *dashboard* que também foi construído usando tons do grupo de paletas *Spectral*.

O grande desafio na elaboração desse gráfico foi incluir no repertório mais uma linguagem de programação, a *Javascript,* pois a dinâmica do gráfico após a geração do arquivo html é comandada por essa linguagem no *backend* do *browser*. Então foi necessário incluir nos parâmetros do *CheckboxGroup* um *call-back* com o código *javascript* que operacionalize a dinâmica de alteração do gráfico após a seleção de quartos na caixa de seleção.

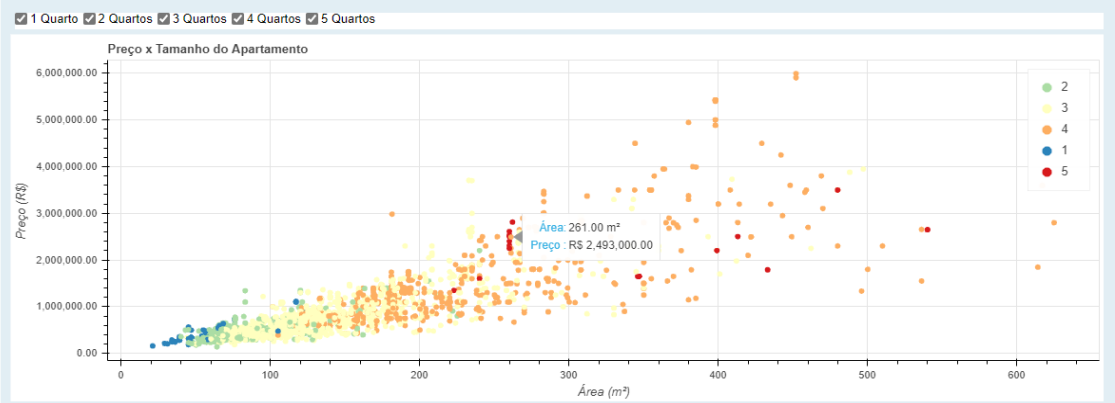
O código pode ser visualizado abaixo.

Figura 78 - Código para criação do gráfico principal

```
#Gráfico Principal

data2 = dict(quartos = Dados_viz.quartos.astype('int').astype('string'),
x = Dados_viz.area,
y = Dados_viz.preco.astype('float'),
z = Dados_viz.quartos.astype('int').astype('string')   )

data2 = pd.DataFrame(data2)

data_source2 = ColumnDataSource(data2)
source2 = ColumnDataSource(dict(x = Dados_viz.area.tolist(),
                                y = Dados_viz.preco.tolist(),
                                z = Dados_viz.quartos.astype('int').astype('string').tolist()))
label2=sorted(['1','2','3','4','5','6'],key=int)

plot2 = figure(plot_width=1150, plot_height=400,tools="hover",
               tooltips=[("Área", "@{x}{0.2f} m²"),("Preço ","R$ @{y}{,0.2f}")],
               toolbar_location=None, title="Preço x Tamanho do Apartamento")
plot2.circle('x', 'y', line_width = 2, source = source2, color=factor_cmap("z", Spectral5, label2), legend_field='z')
plot2.yaxis[0].formatter = NumeralTickFormatter(format="0,0.00")
plot2.xaxis.axis_label = "Área (m²)"
plot2.yaxis.axis_label = "Preço (R$)"

callback = CustomJS(args = {'source': source2, 'data_source': data_source2},
code = """
var data = data_source.data;
var s_data = source.data;
var quartos = data['quartos'];
var select_vals = cb_obj.active;
console.log(select_vals);
var x_data = data['x'];
var y_data = data['y'];
var z_data = data['z'];
var x = s_data['x'];
x.length = 0;
var y = s_data['y'];
y.length = 0;
var z = s_data['z'];
z.length = 0;
for (var i = 0; i < x_data.length; i++) {
    if (select_vals.indexOf(quartos[i]-1) >= 0) {
        x.push(x_data[i]);
        y.push(y_data[i]);
        z.push(z_data[i]);
        }
}
source.change.emit();
console.log("callback completed");
""")

chkbxgrp2 = CheckboxGroup(inline=True, labels = ['1 Quarto', '2 Quartos', '3 Quartos','4 Quartos','5 Quartos'],
                          active=[0,1,2,3,4,5], background='white', default_size=1145, align='start')
chkbxgrp2.js_on_change('active', callback)


##########################################################
```

Detalhe do gráfico principal.

Figura 79 - Detalhe do gráfico principal com hover acionado

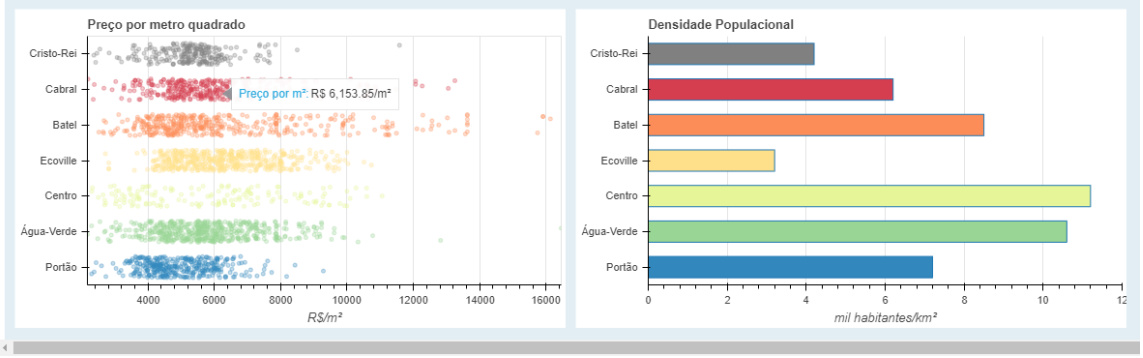


Os gráficos de barras horizontais foram apresentados com o intuito de fazer a comparação do preço do metro quadrado dos apartamentos com a densidade populacional dos bairros de Curitiba-PR. Nesse caso, o usuário pode verificar em qual faixa de preço/m² estão os apartamentos em cada bairro e ver se a concentração de pessoas naquele bairro é um fator que pode influenciar no preço. Também pode verificar em quais bairros há uma dispersão maior de valores, como no bairro Batel onde há variação no preço/m² que vai desde menos de R$ 4.000,00/m² até cerca de R$ 16.000,00/m². Contam com um *hover* com informações numéricas sobre os gráficos, dando maior interatividade com o usuário. Os códigos podem ser visualizados a seguir.

Figura 80 - Código para criação dos gráficos de barras horizontais

```
##################################################
#Gráfico Barra Horizontal 1 - Preço/m²

Bairros = Dados_viz.bairro.unique().tolist()
source3 = ColumnDataSource(Dados_viz)
plot3 = figure(plot_width=570, plot_height=330, y_range=Bairros,
           title="Preço por metro quadrado", toolbar_location=None, tools="hover",
               tooltips=[("Preço por m²", "R$ @{preco_metro}{,0.2f}/m²")])
plot3.circle(x='preco_metro', y=jitter('bairro', width=0.6, range=plot3.y_range),
             source=source3, alpha=0.3, color=factor_cmap("bairro", Spectral6, Bairros))


plot3.x_range.range_padding = 0
plot3.xaxis.axis_label = "R$/m²"
plot3.ygrid.grid_line_color = None



##################################################

#Gráfico Barra Horizontal 2 - Densidade Populacional

Bairros = Dados_viz.bairro.unique().tolist()
group = Dados_viz.groupby('bairro')
source4 = ColumnDataSource(group)

plot4 = figure(y_range=Bairros, x_range=(0,12), plot_width=570, plot_height=330, toolbar_location=None,
           title="Densidade Populacional", tools="hover",
               tooltips=[("Densidade populacional", "@{Densidade_populacional_max}{0.2f} hab/km²")])
plot4.hbar(y="bairro", left=0, right='Densidade_populacional_max', height=0.6,
           source=source4, fill_alpha=1., fill_color=factor_cmap("bairro", Spectral6, Bairros))

plot4.ygrid.grid_line_color = None
plot4.xaxis.axis_label = "mil habitantes/km²"
plot4.outline_line_color = None
```

Detalhe dos gráficos de barras horizontais com *hover acionado.*



Os gráficos de barras verticais empilhadas têm por finalidade apresentar outras grandes variáveis que impactam no valor dos imóveis, quais sejam quartos, suítes e vagas de estacionamento. Todos foram elaborados com paletas do grupo *Spectral* e apresentam no eixo “y” a contagem de apartamentos de acordo com a quantidade de itens (quartos, suítes, vagas). Também possuem um *hover* com informações numéricas como números de quartos (auxiliando a legenda) e quantidade de unidades (apartamentos). Para agregar mais informação no gráfico, foram segregadas as colunas por bairros para melhor visualização da distinção dos portes de apartamentos em cada um deles, ou seja, em quais bairros estão os maiores apartamento, com mais suítes, etc. O código em python segue abaixo:

Figura 81 – Código para geração do gráfico de barras vertical empilhado 1 - quartos

```
#Gráfico Barra Vertical Empilhada 1 - Quartos

bairros = sorted(Dados_viz['bairro'].unique().to_list())
quartos = sorted(list(Dados_viz['quartos'].astype('int').astype('string').unique()), key=int)

quartos_count = Dados_viz[['quartos','bairro','preco']]
quartos_count['quartos'] = quartos_count['quartos'].astype('int')
quartos_count = pd.pivot_table(quartos_count, values="preco", index=['bairro'],
                               columns=["quartos"], aggfunc=np.count_nonzero)
quartos_count = quartos_count.fillna(value=0)

k0='bairros'
k1='1'
k2='2'
k3='3'
k4='4'
k5='5'



v0=quartos_count.index.to_list()
v1=quartos_count[1].to_list()
v2=quartos_count[2].to_list()
v3=quartos_count[3].to_list()
v4=quartos_count[4].to_list()
v5=quartos_count[5].to_list()


dado={k0:v0,
    k1:v1,
     k2:v2,
     k3:v3,
     k4:v4,
     k5:v5
   }
dado

source5 = ColumnDataSource(data=dado)

plot5 = figure(x_range=bairros, plot_width=700, plot_height=250, title="Contagem de apartamentos por número de quartos",
           toolbar_location=None, tools="hover", tooltips="$name quartos: @$name unidades")

plot5.vbar_stack(quartos, x='bairros', width=0.9, color=Spectral5, source=source5,#pode ser colunadatasource,
             legend_label=quartos)

plot5.y_range.start = 0
plot5.y_range.end = 800
plot5.x_range.range_padding = 0.1
plot5.xgrid.grid_line_color = None
plot5.axis.minor_tick_line_color = None
plot5.outline_line_color = None
plot5.legend.location = "top_left"
plot5.legend.orientation = "horizontal"
```

Figura 82 - Código para geração do gráfico de barras vertical empilhado 2 - suítes

```
#Gráfico Barra Vertical Empilhada 2 - Suítes

bairros = sorted(Dados_viz['bairro'].unique().to_list())
suites = sorted(list(Dados_viz['suites'].astype('int').astype('string').unique()), key=int)

suites_count = Dados_viz[['suites','bairro','preco']]
suites_count['suites'] = suites_count['suites'].astype('int')
suites_count = pd.pivot_table(suites_count, values="preco", index=['bairro'],
                              columns=["suites"], aggfunc=np.count_nonzero)
suites_count = suites_count.fillna(value=0)

k0='bairros'
k1='1'
k2='2'
k3='3'
k4='4'



v0=suites_count.index.to_list()
v1=suites_count[1].to_list()
v2=suites_count[2].to_list()
v3=suites_count[3].to_list()
v4=suites_count[4].to_list()



dado={k0:v0,
    k1:v1,
     k2:v2,
     k3:v3,
     k4:v4
     }
dado

source6 = ColumnDataSource(data=dado)

plot6 = figure(x_range=bairros, plot_width=700, plot_height=250, title="Contagem de apartamentos por número de suítes",
           toolbar_location=None, tools="hover", tooltips="$name suites: @$name unidades")

plot6.vbar_stack(suites, x='bairros', width=0.9, color=Spectral4, source=source6,#pode ser colunadatasource,
             legend_label=suites)

plot6.y_range.start = 0
plot6.y_range.end = 800
plot6.x_range.range_padding = 0.1
plot6.xgrid.grid_line_color = None
plot6.axis.minor_tick_line_color = None
plot6.outline_line_color = None
plot6.legend.location = "top_left"
plot6.legend.orientation = "horizontal"
```

Figura 83 - Código para criação do gráfico de barras vertical empilhado 3 - vagas de estacionamento

```
#Gráfico Barra Vertical Empilhada 3 - Vagas

bairros = sorted(Dados_viz['bairro'].unique().to_list())
vagas = sorted(list(Dados_viz['vagas'].astype('int').astype('string').unique()), key=int)

vagas_count = Dados_viz[['vagas','bairro','preco']]
vagas_count['vagas'] = vagas_count['vagas'].astype('int')
vagas_count = pd.pivot_table(vagas_count, values="preco", index=['bairro'],
                             columns=["vagas"], aggfunc=np.count_nonzero)
vagas_count = vagas_count.fillna(value=0)

k0='bairros'
k1='1'
k2='2'
k3='3'
k4='4'
k5='5'


v0=vagas_count.index.to_list()
v1=vagas_count[1].to_list()
v2=vagas_count[2].to_list()
v3=vagas_count[3].to_list()
v4=vagas_count[4].to_list()
v5=vagas_count[5].to_list()

dado={k0:v0,
    k1:v1,
     k2:v2,
     k3:v3,
     k4:v4,
     k5:v5

     }
dado

source7 = ColumnDataSource(data=dado)

plot7 = figure(x_range=bairros, plot_width=700, plot_height=250, title="Contagem de apartamentos por número de vagas",
          toolbar_location=None, tools="hover", tooltips="$name vagas: @$name unidades")

plot7.vbar_stack(vagas, x='bairros', width=0.9, color=Spectral5, source=source7,#pode ser colunadatasource,
            legend_label=vagas)

plot7.y_range.start = 0
plot7.y_range.end = 800
plot7.x_range.range_padding = 0.1
plot7.xgrid.grid_line_color = None
plot7.axis.minor_tick_line_color = None
plot7.outline_line_color = None
plot7.legend.location = "top_left"
plot7.legend.orientation = "horizontal"
```

Detalhe dos gráficos de barras verticais empilhadas com *hover acionado.*

Figura 84 - Gráficos de barras verticais empilhadas com hover acionado

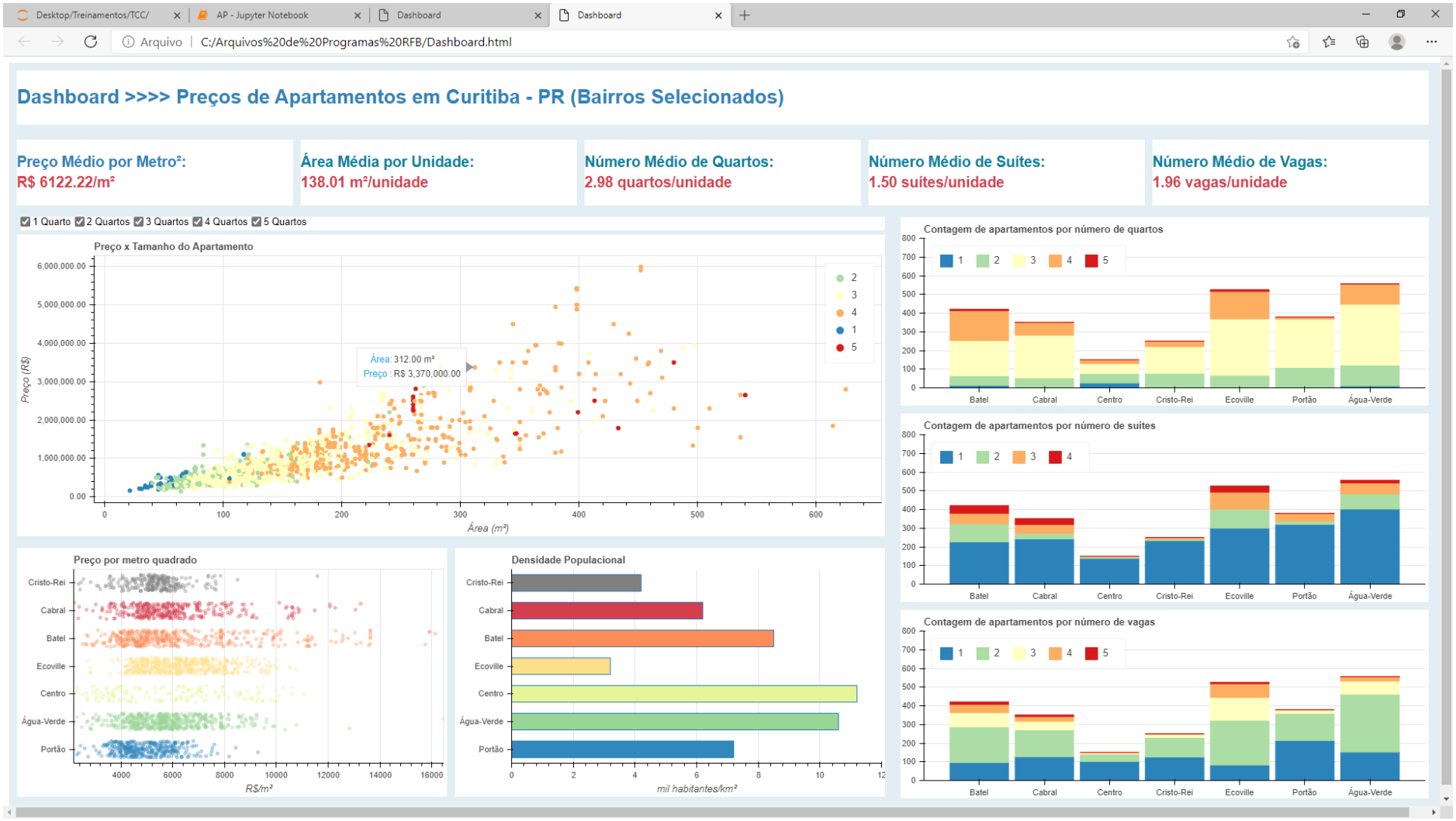
Por fim, temos os códigos para a construção do *layout* do *dashboard* com o encadeamento de colunas e linhas, bem como a exportação do arquivo .html e a abertura dele na página do navegador.

Figura 85 - Código para configuração do layout, geração do arquivo .html e abertura deste no navegador

```
#configuração do Layout e geração do arquivo .html

layout = Column(Row(heading,margin=(5,5,5,5)),
                Row(kpi,kpi1,kpi2, kpi3, kpi4,  margin=(5,5,5,5)),
                Row(
                    Column(
                        Row(Column(
                            chkbxgrp2,
                            plot2,margin=(0,5,5,5))),

                        Row(Column(plot3,margin=(5,5,5,0)),
                            Column(plot4,margin=(5,5,5,5)),
                            margin=(5,5,5,5)
                           ),margin=(0,5,5,5)),
                    Column(
                        Row(plot5, margin=(5,5,5,5)),
                        Row(plot6, margin=(5,5,5,5)),
                        Row(plot7, margin=(5,5,5,5)),
                          )
                ),

                css_classes=['myclass'],  background='#e2eef4', width=1920
               )

output_file("Dashboard.html", title="Dashboard")


show(layout)
```

E na sequência temos o *dashboard* completo e ampliado para melhor visualização, lembrando que esse *dashboard* consta dos anexos deste TCC, no repositório cujo link será informado no próximo tópico, onde é possível interagir com ele e explorar melhor os dados.

Figura 86 - Dashboard completo com hover acionado no gráfico principal



7. Links

A seguir estão os links para o repositório deste trabalho e para o vídeo explicativo.

- Link para o vídeo: https://youtu.be/vySINCDqhaU

- Link para o repositório dos arquivos: https://github.com/Alencar2208/TCC-Ciencia-de-Dados-e-Big-Data-2019---PUC-Minas

APÊNDICE

Programação/Scripts

Anexo I – Jupyter Notebook