



Escrever título (escolher no final)

Universidade Federal da Paraíba - CCEN

Gabriel de Jesus Pereira

30 de novembro de 2024

Índice

1 Resumo	5
2 Introdução	6
3 Objetivos	7
3.1 Objetivo Geral	7
3.2 Objetivos Específicos	7
3.3 Organização do Trabalho	7
4 Recursos Computacionais	8
4.1 Linguagem R	8
4.2 Linguagem Python	9
4.3 Quarto	9
4.4 Web Scraping	10
5 Algoritmos de Aprendizado de Máquina	12
5.1 Árvores de decisão	12
5.2 Métodos Ensemble	16
5.2.1 Bagging	17
5.2.2 Random Forest	18
5.2.3 Boosting Trees	20
5.2.4 Stacked generalization	21
5.2.5 Gradient Boosting	22
5.2.6 Diferentes implementações de Gradient Boosting	24
6 Metodologia	26
6.1 Obtenção dos dados	26
6.2 Análise exploratória de dados	27
6.3 Construção do modelo	28
6.3.1 Etapas de pré-processamento	28
6.3.2 Validação cruzada	29
6.4 Otimização de hiperparâmetros	31
6.4.1 Tree-Structured Parzen Estimator	32
6.4.2 Otimização de hiperparâmetros com optuna	33

6.5	Interpretação dos algoritmos de aprendizagem de máquina	35
6.5.1	Individual Conditional Expectation (ICE)	35
6.5.2	Local interpretable model-agnostic explanations (LIME)	36
6.5.3	SHapley Additive exPlanations (SHAP)	36
7	Resultados	37
7.1	Análise exploratória de dados	37
7.2	Tunagem dos modelos	41
7.2.1	Tunagem da Random Forest	41
7.2.2	Tunagem do Gradient Boosting	43
7.2.3	Tunagem do LGBM	43
7.2.4	Tunagem do XGBoost	46
7.3	Resultados dos modelos	48
7.4	Efeito e importância das variáveis na predição	52
8	Conclusão	53
9	Referências Bibliográficas	54

Lista de Figuras

5.1	Exemplo de estrutura de árvore de regressão. A árvore tem cinco folhas e quatro nós internos.	13
6.1	Visualização de K-Fold com 20 folds.	30
7.1	Quantidade de valores ausentes por variáveis	38
7.2	Distribuição das variáveis numéricas.	39
7.3	Comparação entre distribuição dos valores dos imóveis antes e depois da transformação logarítmica.	40
7.4	Gráfico de correlação de Spearman das variáveis independentes.	40
7.5	Resultados da tunagem da Random Forest.	42
7.6	Resultados da tunagem do Gradient Boosting	44
7.7	Resultados da tunagem do LGBM	45
7.8	Resultados da tunagem do XGBoost	47
7.9	Valores previstos em função dos observados do algoritmo Random Forest	49
7.10	Valores previstos em função dos observados do algoritmo Gradient Boosting	49
7.11	Valores previstos em função dos observados do algoritmo Light Gradient Boosting	50
7.12	Valores previstos em função dos observados do algoritmo Extreme Gradient Boosting	51
7.13	Valores previstos em função dos observados do algoritmo Stacking	51

1 Resumo

2 Introdução

3 Objetivos

3.1 Objetivo Geral

3.2 Objetivos Específicos

3.3 Organização do Trabalho

4 Recursos Computacionais

4.1 Linguagem R

R (R CORE TEAM, 2024) é uma linguagem de programação voltada para computação científica e visualização de dados. Seu desenvolvimento foi iniciado pelos professores Ross Ihaka e Robert Gentleman, que a criaram com o objetivo de ser uma linguagem para ensinar introdução à estatística na Universidade de Auckland. A primeira versão do R foi lançada em 1993, e em 1997 ele se tornou oficialmente parte do Projeto GNU.

A linguagem R é fortemente inspirada no paradigma de programação funcional, o que a torna poderosa para manipulação e análise de dados. Em R, as funções são de primeira classe, ou seja, podem ser passadas como argumentos para outras funções, retornadas como resultados e atribuídas a variáveis. Isso permite a criação de códigos mais modulares e reutilizáveis, além de facilitar a implementação de pipelines de transformação de dados.

O R oferece funções como `apply`, `lapply`, `sapply`, além de pacotes como `purrr` e `furrr`, que são voltados para programação funcional. Esses recursos permitem a aplicação de funções em estruturas de dados de maneira concisa e eficiente, simplificando o processamento e a transformação de grandes conjuntos de dados. Embora o R seja fortemente inspirado pelo paradigma funcional, o R também oferece suporte à programação orientada a objetos, com sistemas como S3, S4 e R6.

Neste trabalho, a linguagem R foi utilizada para a raspagem de dados de imóveis e para a geocodificação dos endereços desses imóveis. O R possui um vasto ecossistema de pacotes focados na coleta de dados da web. Para a raspagem, foram utilizados os pacotes `xml2` (WICKHAM; HESTER; OOMS, 2023), `rvest` (WICKHAM, 2024) e `httr` (WICKHAM, 2023) para fazer as requisições HTTP, além do Selenium (THORPE, 2024), que permitiu interagir com conteúdos dinâmicos das páginas. A obtenção das coordenadas dos imóveis foi realizada através da geocodificação de seus endereços, utilizando a biblioteca `tidygeocoder` (CAMBON *et al.*, 2021).

4.2 Linguagem Python

Python (VAN ROSSUM; DRAKE JR, 1995) é uma linguagem de programação criada por Guido Van Rossum. Guido começou a desenvolver o Python no final dos anos 80 como uma sucessora da linguagem ABC, e a primeira versão foi lançada oficialmente em 1991. Python é uma linguagem de alto nível, de propósito geral, que enfatiza a simplicidade e a legibilidade do código.

Neste trabalho, Python foi utilizado principalmente para modelagem e coleta de dados de sites de imóveis. Para a modelagem, utilizou-se a biblioteca scikit-learn (PEDREGOSA *et al.*, 2011), que oferece uma grande quantidade de ferramentas estatísticas e de aprendizado de máquina. A otimização dos hiperparâmetros dos modelos foi realizada com a biblioteca Optuna, que dispõe de diversos métodos de otimização, incluindo métodos avançados de otimização bayesiana. Para a coleta de dados de imóveis, foi empregada a biblioteca Scrapy (KOUZIS-LOUKAS, 2016), um framework open-source em Python voltado para web scraping, que permite extrair dados de páginas da web de forma automatizada. Com o Scrapy, é possível criar “crawlers” ou “spiders” que navegam em sites, coletando e organizando informações específicas de maneira eficiente. Além do Scrapy, foi utilizado também a biblioteca Playwright, com o objetivo de interagir com o conteúdo dinâmico presente na página.

Além disso, Python foi utilizado para a criação de todos os gráficos e para a manipulação das bases de dados. Para visualização, foram usadas as bibliotecas Matplotlib (HUNTER, 2007) e Seaborn (WASKOM, 2021), enquanto a manipulação e organização das bases de dados foi feita com a biblioteca Pandas (MCKINNEY, 2010; TEAM, 2020), amplamente utilizada em Python para limpeza, análise e organização de dados, além de suportar operações estatísticas.

4.3 Quarto

Quarto (ALLAIRE; DERVIEUX, 2024) é uma plataforma de publicação científica desenvolvida pela empresa Posit com o objetivo de criar documentos de alta qualidade a partir de arquivos que combinam texto e código. É uma evolução do R Markdown, sendo compatível com várias linguagens de programação, como R, Python, Julia e outros, o que o torna bastante versátil para a análise de dados e geração de relatórios interativos. Com o Quarto, é possível produzir relatórios, artigos, livros, apresentações e sites. Ele é amplamente utilizado na comunidade R e oferece suporte para Markdown e L^AT_EX, facilitando a inclusão de fórmulas matemáticas, gráficos, tabelas e outros elementos visuais, que são renderizados em formatos como HTML, PDF e MS Word.

O Quarto foi utilizado para a produção de todo o texto deste trabalho, atendendo

aos requisitos das normas ABNT. Com sua flexibilidade e suporte para formatação customizável, Quarto permitiu integrar texto, código e visualizações de forma reproduzível e organizada, facilitando a geração de documentos acadêmicos de alta qualidade.

4.4 Web Scraping

Web scraping, ou raspagem de dados, é uma técnica utilizada para extrair informações de sites na internet, salvando-as em arquivos ou sistemas de banco de dados para realizar análise, construção de aplicações ou ter acesso a informações de difícil disponibilização. Geralmente a raspagem de dados é realizada utilizando o Hypertext Transfer Protocol (HTTP) ou a partir da simulação do comportamento de um usuário de navegador. O HTTP é o protocolo responsável por fazer toda a comunicação cliente-servidor contida na internet com base na definição de oito métodos de requisição: GET, HEAD, POST, PUT, DELETE, TRACE, OPTIONS e CONNECT. Cada método indica a ação a ser realizada no recurso especificado.

- GET: O método GET serve para requisitar uma representação do recurso especificado. Ou seja, ele serve para visualizar dados de um site.
- HEAD: O HEAD é bastante semelhante o GET, mas ele retorna apenas metadados sobre um recurso no servidor, sem que o recurso seja retornado. Ele retorna todos os cabeçalhos associados a um recurso em uma determinada URL.
- POST: O método POST envia dados para serem processados para o recurso especificado. Esses dados podem ser, por exemplo, dados de um formulário HTML.
- PUT: O PUT é bastante semelhante ao POST, ele envia os dados de forma semelhante. No entanto, caso seja necessário atualizar um usuário diversas vezes, o método PUT vai sobrescrever os dados e ficará apenas um único registro atualizado. Para o método POST, serão criados diversos registros para cada requisição realizada.
- DELETE: Exclui o recurso.
- TRACE: O método TRACE HTTP é usado para diagnóstico, depuração e solução de problemas. Ele simplesmente retorna um rastreamento de diagnóstico que registra dados do ciclo de forma que o cliente possa saber o que os servidores intermediários estão mudando em sua requisição.
- OPTIONS: O método OPTIONS retorna uma lista de quais métodos HTTP são suportados e permitidos pelo servidor.

- CONNECT: O CONNECT é usado para criar uma conexão com um recurso do lado do servidor. O alvo mais comum do método HTTP CONNECT é um servidor proxy, que um cliente deve acessar para sair da rede local.

Toda raspagem inicia com a composição de uma requisição HTTP para adquirir recursos de um site. Geralmente, essa requisição é formatada numa consulta GET ou em uma mensagem HTTP contendo uma consulta POST. Quando a requisição é recebida e processado com sucesso, o recurso requisitado é salvo site e enviado de volta em diversos formatos de arquivos como, HTML, XML, JSON ou dados multímidia. Após o download do conteúdo do site, o processo de extração continua com a reformatação e organização dos dados de forma estruturada.

A extração de informações de um arquivo HTML é geralmente feita com módulos como BeautifulSoup (RICHARDSON, 2007), em Python, ou rvest, em R. No entanto, muitas vezes é necessário interagir com conteúdos dinâmicos da página para carregar todas as informações disponíveis. Para esses casos, são utilizadas bibliotecas como Playwright ou Selenium, que permitem lidar com autenticação, cookies e redirecionamentos, simulando a navegação em um navegador como Google Chrome, Firefox, ou outros. Essas bibliotecas fazem requisições HTTP enquanto reproduzem a experiência de um usuário real, com o objetivo de automatizar processos de busca e interação com a página.

A raspagem de dados possui diversas utilidades, como monitoramento de variações de preços, análise de produtos, acompanhamento de condições tempo, análise da oscilação de preços de ativos do mercado financeiro ao longo do tempo ou coleta de postagens em redes sociais para investigação de opinião pública. Neste trabalho, a raspagem de dados foi utilizada para coletar informações sobre imóveis na cidade de João Pessoa, capital da Paraíba.

5 Algoritmos de Aprendizado de Máquina

Neste capítulo, serão descritos os algoritmos de aprendizado de máquina utilizados neste trabalho. Alguns dos métodos utilizados podem fazer uso de diversos algoritmos ou modelos estatísticos. No entanto, o foco principal e o mais utilizado foram as árvores de decisão, especialmente em sua forma particular, as árvores de regressão. Assim, os algoritmos descritos são métodos baseados em árvores.

Os métodos baseados em árvore envolvem a estratificação ou segmentação do espaço dos preditores¹ em várias regiões simples. Dessa forma, todos os algoritmos utilizados neste trabalho partem dessa ideia. Portanto, o primeiro a ser explicado será o de árvores de decisão, pois fundamenta todos os outros algoritmos. Depois das árvores de decisão, serão explicados os métodos ensemble e, por fim, diferentes variações do método de gradient boosting.

5.1 Árvores de decisão

Árvores de decisão podem ser utilizadas tanto para regressão quanto para classificação. Elas servem de base para os modelos baseados em árvores empregados neste trabalho, focando particularmente nas árvores de regressão². O processo de construção de uma árvore se baseia no particionamento recursivo do espaço dos preditores, onde cada particionamento é chamado de nó e o resultado final é chamado de folha ou nó terminal. Em cada nó, é definida uma condição e, caso essa condição seja satisfeita, o resultado será uma das folhas desse nó. Caso contrário, o processo segue para o próximo nó e verifica a próxima condição, podendo gerar uma folha ou outro nó. Veja um exemplo na Figura 5.1.

O espaço dos preditores é dividido em J regiões distintas e disjuntas denotadas por R_1, R_2, \dots, R_J . Essas regiões são construídas em formato de caixa de forma a minimizar a soma dos quadrados dos resíduos. Dessa forma, pode-se modelar a variável resposta como uma constante c_j em cada região R_j .

¹O espaço dos preditores é o conjunto de todos os valores possíveis para as variáveis independentes \mathbf{x}

²Uma árvore de regressão é um caso específico da árvore de decisão, mas para regressão.

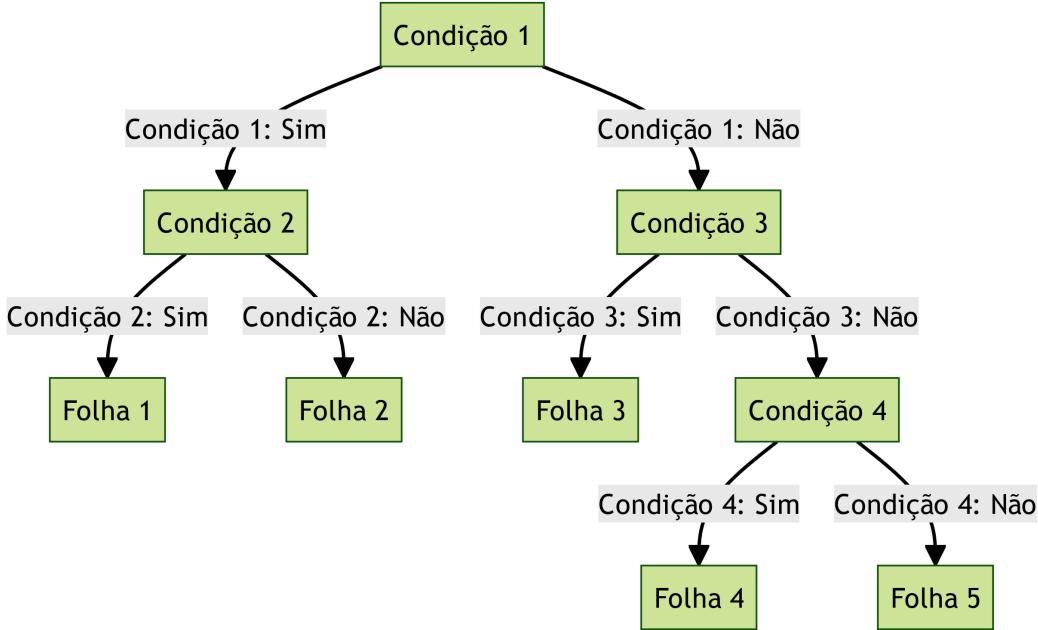


Figura 5.1: Exemplo de estrutura de árvore de regressão. A árvore tem cinco folhas e quatro nós internos.

$$f(x) = \sum_{j=1}^J c_j I(x \in R_j)$$

O estimador para a constante c_j é encontrado pelo método de mínimos quadrados. Assim, deve-se minimizar $\sum_{x_i \in R_j} [y_i - f(x_i)]^2$. No entanto, perceba que $f(x_i)$ está sendo avaliado somente em um ponto específico x_i , o que reduzirá $f(x_i)$ para uma constante c_j . É fácil de se chegar ao resultado se for observada a definição da função indicadora $I(x \in R_j)$

$$I_{R_j}(x_i) = \begin{cases} 1, & \text{se } x_i \in R_j \\ 0, & \text{se } x_i \notin R_j \end{cases}$$

Como as regiões são disjuntas, x_i não pode estar simultaneamente em duas regiões. Assim, para um ponto específico x_i , apenas um dos casos da função indicadora será diferente de 0. Portanto, $f(x_i) = c_j$. Agora, derivando $\sum_{x_i \in R_j} (y_i - c_j)^2$ em relação a c_j

$$\frac{\partial}{\partial c_j} \sum_{x_i \in R_j} (y_i - c_j)^2 = -2 \sum_{x_i \in R_j} (y_i - c_j) \quad (5.1)$$

e igualando Equação 5.1 a 0, tem-se a seguinte igualdade

$$\sum_{x_i \in R_j} (y_i - \hat{c}_j) = 0$$

que se abrirmos o somatório e dividirmos pelo número total de pontos N_j na região R_j , teremos que o estimador de c_j será simplesmente a média dos y_i na região R_j :

$$\sum_{x_i \in R_j} y_i - \hat{c}_j N_j = 0 \Rightarrow \hat{c}_j = \frac{1}{N_j} \sum_{x_i \in R_j} y_i \quad (5.2)$$

No entanto, JAMES *et al.* (2013) caracteriza como inviável considerar todas as possíveis partições do espaço das variáveis em J caixas devido ao alto custo computacional. Dessa forma, a abordagem a ser adotada é uma divisão binária recursiva. O processo começa no topo da árvore de regressão, o ponto em que contém todas as observações, e continua sucessivamente dividindo o espaço dos preditores. As divisões são indicadas como dois novos ramos na árvore, como pode ser visto na Figura 5.1.

Para executar a divisão binária recursiva, deve-se primeiramente selecionar a variável independente X_j e o ponto de corte s tal que a divisão do espaço dos preditores conduza a maior redução possível na soma dos quadrados dos resíduos. Dessa forma, definimos dois semi-planos

$$R_1(j, s) = \{X | X_j \leq s\} \text{ e } R_2(j, s) = \{X | X_j > s\}$$

e procuramos a divisão da variável j e o ponto de corte s que resolve a equação

$$\min_{j,s} \left[\min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2 \right]$$

em que c_1 e c_2 é a média da variável dependente para as observações de treinamento nas regiões $R_1(j, s)$ e $R_2(j, s)$, respectivamente. Assim, encontrando a melhor divisão, os dados são particionados nas duas regiões resultantes e o processo de divisão é repetido em todas as outras regiões.

O tamanho da árvore pode ser considerado um hiperparâmetro para regular a complexidade do modelo, pois uma árvore muito grande pode causar sobreajuste aos dados de treinamento, capturando não apenas os padrões relevantes, mas também o ruído. Como resultado, o modelo pode apresentar bom desempenho nos dados de treinamento, mas falhar ao lidar com novos dados devido à sua incapacidade de generalização. Por outro lado, uma árvore muito pequena pode não captar padrões,

relações e estruturas importantes presentes nos dados. Dessa forma, a estratégia adotada para selecionar o tamanho da árvore consiste em crescer uma grande árvore T_0 , interrompendo o processo de divisão apenas ao atingir um tamanho mínimo de nós. Posteriormente, a árvore T_0 é podada utilizando o critério de custo complexidade, que será definido a seguir.

Para o processo de poda da árvore, definimos uma árvore qualquer T que pode ser obtida através do processo da poda de T_0 , de modo que $T \subset T_0$. Assim, sendo N_j a quantidade de pontos na região R_j , seja

$$Q_j(T) = \frac{1}{N_j} \sum_{x_i \in R_j} (y_i - \hat{c}_j)^2$$

uma medida de impureza do nó pelo erro quadrático médio. Assim, define-se o critério de custo complexidade

$$C_\alpha(T) = \sum_{m=1}^{|T|} N_j Q_j(T) + \alpha |T|$$

onde $|T|$ denota a quantidade total de folhas, e $\alpha \geq 0$ é um hiperparâmetro que equilibra o tamanho da árvore e a adequação aos dados. A ideia é encontrar, para cada α , a árvore $T_\alpha \subset T_0$ que minimiza $C_\alpha(T)$. Valores grandes de α resultam em árvores menores, enquanto valores menores resultam em árvores maiores, e $\alpha = 0$ resulta na própria árvore T_0 . A busca por T_α envolve colapsar sucessivamente o nó interno que provoca o menor aumento em $\sum_j N_j Q_j(T)$, continuando o processo até produzir uma árvore com um único nó. Esse processo gera uma sequência de subárvores, na qual existe uma única subárvore menor que, para cada α , minimiza $C_\alpha(T)$.

A estimativa de α é realizada por validação cruzada com cinco ou dez folds, sendo $\hat{\alpha}$ escolhido para minimizar a soma dos quadrados dos resíduos durante o processo de validação cruzada. Assim, a árvore final será $T_{\hat{\alpha}}$. O Algoritmo 5.1 exemplifica o processo de crescimento de uma árvore de regressão:

No caso de uma árvore de decisão para classificação, a principal diferença está no critério de divisão dos nós e na poda da árvore. Para a classificação, a previsão em um nó j , correspondente a uma região R_j com N_j observações, será simplesmente a classe majoritária. Assim, tem-se

$$\hat{p}_{jk} = \frac{1}{N_j} \sum_{x_i \in R_j} I(y_i = k)$$

como a proporção de observações da classe k no nó j . Dessa forma, as observações no nó j são classificadas na classe $k(j) = \arg \max_k \hat{p}_{jk}$, que é a moda no nó j .

Algoritmo 5.1 Algoritmo para crescer uma árvore de regressão

1. Use a divisão binária recursiva para crescer uma árvore grande T_0 nos dados de treinamento, parando apenas quando cada folha tiver menos do que um número mínimo de observações.
 2. Aplique o critério custo de complexidade à árvore grande T_0 para obter uma sequência de melhores subárvore T_α , em função de α .
 3. Use validação cruzada K -fold para escolher α . Isto é, divida as observações de treinamento em K folds. Para cada $k = 1, \dots, K$:
 - (a) Repita os Passos 1 e 2 em todos os folds, exceto no k -ésimo fold dos dados de treinamento.
 - (b) Avalie o erro quadrático médio de previsão nos dados no k -ésimo fold deixado de fora, em função de α . Faça a média dos resultados para cada valor de α e escolha α que minimize o erro médio.
 4. Retorne a subárvore $T_{\hat{\alpha}}$ do Passo 2 que corresponde ao valor estimado de α .
-

Algoritmo 5.1: Fonte: JAMES *et al.* (2013, p. 337).

Para a divisão dos nós no caso da regressão, foi utilizado o erro quadrático médio como medida de impureza. Para a classificação, algumas medidas comuns para $Q_j(T)$ são o erro de classificação, o índice de Gini ou a entropia cruzada.

5.2 Métodos Ensemble

As árvores de decisão são conhecidas por sua alta interpretabilidade, mas geralmente apresentam um desempenho preditivo inferior em comparação com outros modelos e algoritmos. No entanto, é possível superar essa limitação construindo um modelo preditivo que combina a força de uma coleção de estimadores base, um processo conhecido como aprendizado em conjunto (Ensemble Learning). De acordo com HASTIE *et al.* (2009), o aprendizado em conjunto pode ser dividido em duas etapas principais: a primeira etapa consiste em desenvolver uma população de algoritmos de aprendizado base a partir dos dados de treinamento, e a segunda etapa envolve a combinação desses algoritmos para formar um estimador agregado. Portanto, nesta seção, serão definidos os métodos de aprendizado em conjunto utilizados neste trabalho.

5.2.1 Bagging

O algoritmo de Bootstrap Aggregation, ou Bagging, foi introduzido por BREIMAN (1996). Sua ideia principal é gerar um estimador agregado a partir de múltiplas versões de um preditor, que são criadas por meio de amostras bootstrap do conjunto de treinamento, utilizadas como novos conjuntos de treinamento. O Bagging pode ser empregado para melhorar a estabilidade e a precisão de modelos ou algoritmos de aprendizado de máquina, além de reduzir a variância e evitar o sobreajuste. Por exemplo, o Bagging pode ser utilizado para melhorar o desempenho da árvore de regressão descrita anteriormente.

BREIMAN (1996) define formalmente o algoritmo de Bagging, que utiliza um conjunto de treinamento \mathcal{L} . A partir desse conjunto, são geradas amostras bootstrap $\mathcal{L}^{(B)}$ com B réplicas, formando uma coleção de modelos $\{f(x, \mathcal{L}^{(B)})\}$, onde f representa um modelo estatístico ou algoritmo treinado nas amostras bootstrap para prever ou classificar uma variável dependente y com base em variáveis independentes \mathbf{x} . Se a variável dependente y for numérica, a predição é obtida pela média das previsões dos modelos:

$$f_B(x) = \frac{1}{B} \sum_{b=1}^B f(x, \mathcal{L}^{(b)})$$

onde f_B representa a predição agregada. No caso em que y prediz uma classe, utiliza-se a votação majoritária. Ou seja, se estivermos classificando em classes $j \in 1, \dots, J$, então $N_j = \#\{B; f(x, \mathcal{L}^{(b)}) = j\}$ representa o número de vezes que a classe j foi predita pelos estimadores. Assim,

$$f_B(x) = \arg \max_j N_j$$

isto é, o j para o qual N_j é máximo

Embora a técnica de Bagging possa melhorar o desempenho de uma árvore de regressão ou de classificação, isso geralmente vem ao custo de menor interpretabilidade. Quando o Bagging é aplicado a uma árvore de regressão, construímos B árvores de regressão usando B réplicas de amostras bootstrap e tomamos a média das predições resultantes (JAMES *et al.*, 2013). Nesse processo, as árvores de regressão crescem até seu máximo, sem passar pelo processo de poda, resultando em cada árvore individual com alta variância e baixo viés. No entanto, ao agregar as predições das B árvores, a variância é reduzida.

Para mitigar a falta de interpretabilidade do método Bagging aplicado a árvores de regressão, pode-se usar a medida de impureza baseada no erro quadrático médio,

definida anteriormente, como uma métrica de importância das variáveis independentes. Um valor elevado na redução total média do erro quadrático médio, calculado com base nas divisões realizadas por um determinado preditor em todas as B árvores, indica que o preditor é importante.

As árvores construídas pelo algoritmo de árvore de decisão se beneficiam da proposta de agregação do Bagging, mas esse benefício é limitado devido à correlação positiva existente entre as árvores. Se as árvores forem variáveis aleatórias independentes e identicamente distribuídas, cada uma com variância σ^2 , a variância da média das previsões das B árvores será $\frac{1}{B}\sigma^2$. No entanto, se as árvores forem apenas identicamente distribuídas, mas não necessariamente independentes, e apresentarem uma correlação positiva ρ , a esperança da média das B árvores será a mesma que a esperança de uma árvore individual. Portanto, o viés do agregado das árvores é o mesmo das árvores individuais, e a melhoria é alcançada apenas pela redução da variância. A variância da média das previsões será dada por:

$$\rho\sigma^2 + \frac{1 - \rho}{B}\sigma^2 \quad (5.3)$$

Isso significa que, à medida que o número de árvores B aumenta, o segundo termo da soma se torna menos significativo. Portanto, os benefícios da agregação proporcionados pelo algoritmo de Bagging são limitados pela correlação entre as árvores (HASTIE *et al.*, 2009). Mesmo com o aumento do número de árvores no Bagging, a correlação entre elas impede que as previsões individuais sejam completamente independentes, resultando em menor diminuição da variância da média das previsões do que seria esperado se as árvores fossem totalmente independentes. Uma maneira de melhorar o algoritmo de Bagging é por meio do Random Forest, que será descrito a seguir.

5.2.2 Random Forest

O algoritmo Random Forest é uma técnica derivada do método de Bagging, mas com modificações específicas na construção das árvores. O objetivo é melhorar a redução da variância ao diminuir a correlação entre as árvores, sem aumentar significativamente a variabilidade. Isso é alcançado durante o processo de crescimento das árvores por meio da seleção aleatória de variáveis independentes.

No algoritmo Random Forest, ao construir uma árvore a partir de amostras bootstrap, selecionam-se aleatoriamente $m \leq p$ das p variáveis independentes como candidatas para a divisão, antes de cada ramificação (com $m = p$ no caso do Bagging). Dessa forma, diferente do Bagging, aqui não se considera todas as p variáveis independentes para realizar a divisão e minimizar a impureza, mas apenas m dessas p variáveis. A escolha aleatória de apenas m covariáveis como candidatas para a divisão ajuda a solucionar um

dos principais problemas do algoritmo de Bagging, que tende a gerar árvores de decisão semelhantes, resultando em previsões altamente correlacionadas. O Random Forest busca diminuir esse problema ao criar oportunidades para que diferentes preditores sejam considerados. Em média, uma fração $(p - m)/p$ das divisões nem sequer incluirá o preditor mais forte como candidato, permitindo que outros preditores tenham a chance de serem selecionados (JAMES *et al.*, 2013). Esse mecanismo reduz a correlação entre as árvores, o que, por sua vez, diminui a variabilidade das previsões produzidas pelas árvores.

Algoritmo 5.2 Algoritmo de uma Random Forest para regressão ou classificação

1. Para $b = 1$ até B :

- (a) Construa amostras bootstrap \mathcal{L}^* de tamanho N dos dados de treinamento.
- (b) Faça crescer uma árvore de floresta aleatória T_b para os dados bootstrap, repetindo recursivamente os seguintes passos para cada folha da árvore, até que o tamanho mínimo do nó n_{min} seja atingido.
 - i. Selecione m variáveis aleatoriamente entre as p variáveis.
 - ii. Escolha a melhor variável entre as m .
 - iii. Divida o nó em dois subnós.

2. Por fim, o conjunto de árvores $\{T_b\}_1^B$ é construído.

No caso da regressão, para fazer uma previsão em um novo ponto x , temos a seguinte função:

$$\hat{f}_{rf}^B(x) = \frac{1}{B} \sum_{b=1}^B T_b(x)$$

Para a classificação é utilizado o voto majoritário. Assim, seja $\hat{C}_b(x)$ a previsão da classe da árvore de floresta aleatória b . Então,

$$\hat{C}_{rf}^B(x) = \arg \max_c \sum_{b=1}^B I(\hat{C}_b(x) = c)$$

onde c representa as classes possíveis.

Algoritmo 5.2: Fonte: HASTIE *et al.* (2009, p. 588).

A quantidade de variáveis independentes m selecionadas aleatoriamente é um hiperparâmetro que pode ser estimado por meio de validação cruzada. Valores comuns para m são $m = \sqrt{p}$ com tamanho mínimo do nó igual a um para classificação, e $m = p/3$ com tamanho mínimo do nó igual a cinco para regressão (HASTIE *et al.*, 2009). Quando o número de variáveis é grande, mas poucas são realmente relevantes, o algoritmo Random Forest pode ter um desempenho inferior com valores pequenos de m , pois isso reduz as chances de selecionar as variáveis mais importantes. No entanto, usar um valor pequeno de m pode ser vantajoso quando há muitos preditores correlacionados. Além disso, assim como no Bagging, a Random Forest não sofre de sobreajuste com o aumento da quantidade de árvores B . Portanto, é suficiente usar um B grande o bastante para que a taxa de erro se estabilize (JAMES *et al.*, 2013).

5.2.3 Boosting Trees

O Boosting, assim como o Bagging, é um método destinado a melhorar o desempenho de modelos ou algoritmos. No entanto, neste trabalho, o Boosting foi aplicado apenas às árvores de regressão. Portanto, a explicação do Boosting será restrito ao caso de Boosting Trees (Algoritmo 5.3).

No algoritmo de Bagging, cada árvore é construída e ajustada utilizando amostras bootstrap, e ao final, um estimador agregado φ_B é formado a partir das B árvores. O Boosting Trees funciona de maneira semelhante, mas sem o uso de amostras bootstrap. A ideia principal é corrigir os erros das árvores anteriores, ajustando as novas árvores aos resíduos das anteriores, visando melhorar suas previsões. Assim, as árvores são construídas de forma sequencial, incorporando as informações das árvores anteriores.

No caso da regressão, o Boosting combina um grande número de árvores de decisão $\hat{f}^1, \dots, \hat{f}^B$. A primeira árvore é construída utilizando o conjunto de dados original, e seus resíduos são calculados. Com a primeira árvore ajustada, a segunda árvore é ajustada aos da árvore anterior resíduos e, em seguida, é adicionada ao estimador para atualizar os resíduos. Dessa forma, os resíduos servem como informação crucial para construir novas árvores e corrigir os erros das árvores anteriores. Como cada nova árvore depende das árvores já construídas, árvores menores são suficientes (JAMES *et al.*, 2013).

O processo de aprendizado no método de Boosting é lenta, o que acaba gerando melhores resultados. Esse processo de aprendizado pode ser controlado por um hiperparâmetro λ chamado de shrinkage, ou taxa de aprendizado, permitindo que mais árvores, com formas diferentes, corrijam os erros das árvores passadas. No entanto, um valor muito pequeno para λ requer uma quantidade muito maior B de árvores e, diferente do Bagging e Random Forest, o Boosting pode sofrer de sobreajuste se a quantidade de árvores é muito grande. Além disso, a quantidade de divisões d em

Algoritmo 5.3 Método Boosting aplicado a árvores de regressão

1. Defina $\hat{f}(x) = 0$ e $r_i = y_i$ para todos os i no conjunto de treinamento
2. Para $b = 1, 2, \dots, B$, repita:
 - (a) Ajuste uma árvore \hat{f}^b com d divisões para os dados de treinamento (X, r) .
 - (b) Atualize \hat{f} adicionando uma versão com o hiperparâmetro λ de taxa de aprendizado:

$$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x)$$

- (c) Atualize os resíduos,

$$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i)$$

3. Retorne o modelo de boosting,

$$\hat{f}(x) = \sum_{b=1}^B \lambda \hat{f}^b(x)$$

Algoritmo 5.3: Fonte: JAMES *et al.* (2013, p. 349).

cada árvore, que controla a complexidade do boosting, pode ser considerado também um hiperparâmetro. Para $d = 1$ é ajustado um modelo aditivo, já que cada termo envolve apenas uma variável. JAMES *et al.* (2013) define d como a profundidade de interação que controla a ordem de interação do modelo boosting, já que d divisões podem envolver no máximo d variáveis.

5.2.4 Stacked generalization

A Stacked Generalization, ou Stacking, é um método de ensemble que consiste em treinar um modelo gerado a partir da combinação da predição de vários outros modelos, visando melhorar a precisão das previsões. Esse método pode ser aplicado a qualquer modelo estatístico ou algoritmo de aprendizado de máquina. A ideia principal é atribuir pesos às previsões, de modo a dar maior importância aos modelos que produzem melhores resultados, ao mesmo tempo em que se evita atribuir altos pesos a modelos com alta complexidade.

Matematicamente, o Stacking define previsões $\hat{f}_m^{-i}(x)$ em x , utilizando o modelo m , aplicado ao conjunto de treinamento com a i -ésima observação removida (HASTIE *et al.*, 2009). Assim, os pesos são estimados de forma a minimizar o erro de previsão combinado, dado pela seguinte expressão:

$$\hat{w}^{st} = \arg \min_w \sum_{i=1}^N \left[y_i - \sum_{m=1}^M w_m f_m^{-i}(x_i) \right]^2$$

A previsão final dos modelos empilhados é $\sum_m \hat{w}_m^{st} \hat{f}_m(x)$. Assim, em vez de escolher um único modelo, o método de Stacking combina os modelos utilizando pesos estimados, o que melhora a performance preditiva, mas pode comprometer a interpretabilidade.

5.2.5 Gradient Boosting

O algoritmo de Gradient Boosting é semelhante ao de Boosting, mas com diferenças mínimas. Ele constrói modelos aditivos ajustando sequencialmente funções bases aos pseudos-resíduos, que correspondem aos gradientes da função perda do modelo atual (FRIEDMAN, 2002). Esses gradientes indicam a direção na qual a função perda diminui. Neste trabalho, foram utilizadas diferentes implementações de Gradient Boosting. No entanto, todas empregam o Gradient Boosting com árvores de regressão, com algumas modificações para a construção das árvores ou para melhorar a eficiência do algoritmo existente. Assim, o algoritmo a ser explicado será o Gradient Tree Boosting (Algoritmo 5.4).

O Gradient Boosting aplicado para árvores de regressão, tem que cada função base é uma árvore de regressão com J_m folhas. Dessa forma, cada árvore de regressão tem a forma aditiva

$$h_m(x; \{b_j, R_j\}_1^J) = \sum_{j=1}^{J_m} b_{jm} I(x \in R_{jm}) \quad (5.4)$$

em que $\{R_{jm}\}_1^{J_m}$ são as regiões disjuntas que, coletivamente, cobrem o espaço de todos os valores conjuntos das variáveis preditoras \mathbf{x} . Essas regiões são representadas pelas folhas de sua correspondente árvore. Como as regiões são disjuntas, Equação 5.4 se reduz simplesmente a $h_m(x) = b_{jm}$ para $x \in R_{jm}$. Por mínimos quadrados, b_{jm} é simplesmente a média dos pseudo-resíduos r_{im} ,

$$\hat{b}_{jm} = \frac{1}{N_{jm}} \sum_{x_i \in R_{jm}} r_{im}$$

que dão a direção de diminuição da função perda L pela expressão do gradiente da linha 2(a). Assim, cada árvore de regressão é ajustada aos r_{im} de forma a minimizar o erro

Algoritmo 5.4 Gradient Tree Boosting

1. Inicialize $f_0(x) = \arg \min_{\gamma} \sum_{i=1}^N L(y_i, \gamma)$

2. Para $m = 1$ até M :

(a) Para $i = 1, 2, \dots, N$, calcule

$$r_{im} = - \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right]_{f=f_{m-1}}$$

(b) Ajuste uma árvore de regressão aos pseudo-resíduos r_{im} , obtendo regiões terminais R_{jm} , $j = 1, 2, \dots, J$.

(c) Para $j = 1, 2, \dots, J_m$, calcule

$$\gamma_{jm} = \arg \min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma)$$

(d) Atualize $f_m(x) = f_{m-1}(x) + \lambda \sum_{j=1}^J \gamma_{jm} I(x \in R_{jm})$

3. Retorne $\hat{f}(x) = f_M(x)$

Algoritmo 5.4: Fonte: HASTIE *et al.* (2009, p. 361).

das árvores anteriores. N_{jm} denota a quantidade de pontos na região R_{jm} . Por fim, o estimador é separadamente atualizado em cada região correspondente e é expresso

$$f_m(x) = f_{m-1}(x) + \lambda \sum_{j=1}^J \gamma_{jm} I(x \in R_{jm})$$

em que γ_{jm} representa a atualização da constante ótima para cada região, baseado na função perda L , dada a aproximação $f_{m-1}(x)$. O $0 < \lambda \leq 1$, assim como no algoritmo de boosting, representa o hiperparâmetro shrinkage para controlar a taxa de aprendizado. Pequenos valores de λ necessitam maiores quantidades de iterações M para diminuir o risco de treinamento.

As outras implementações de Gradient Boosting aplicadas à árvores de decisão tem seus próprios motivos de existência. Esses motivos incluem a busca por maior eficiência computacional, adição de recursos e até mesmo maior flexibilidade. As duas outras implementações utilizadas foram o Extreme Gradient Boosting e Light Gradient

Boosting.

O Extreme Gradient Boosting (CHEN; GUESTRIN, 2016) é uma implementação altamente eficiente do algoritmo e flexível de Gradient Boosting aplicado à árvores de decisão. Além disso, é adicionado um novo recurso, técnicas de regularização para diminuir o sobreajuste do modelo. A função objetivo agora passa a ser definida da seguinte forma:

$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

em que l é uma função de perda convexa diferenciável e o segundo termo *Omega* penaliza a complexidade de cada função de árvore de decisão. O termo de regularização adicional ajuda a suavizar os pesos finais para evitar o sobreajuste. Ω é definido dado como:

$$\Omega(f) = \gamma T + \frac{1}{2} \lambda \|\omega\|^2$$

onde T é a quantidade de folhas na árvore, $\|\omega\|^2$ é a soma do quadrado dos pesos associados às folhas. γ e λ são os parâmetros de regularização, em que λ penaliza os pesos das folhas e γ penaliza a quantidade de folhas nas árvores. Além disso, é fácil ver que $\frac{1}{2}\lambda\|\omega\|^2$ representa a penalização L2 (Ridge). Assim, com a regularização aplicada a função objetivo, modelos e funções preditivas serão mais comuns pois simplificarão o modelo.

5.2.6 Diferentes implementações de Gradient Boosting

O algoritmo de Gradient Boosting aplicado a árvores de decisão é um dos mais eficientes e amplamente utilizados, sendo frequentemente vencedor em competições de modelagem estatística. Existem diversas variações desse algoritmo, cada uma com seus próprios motivos de existência. Esses motivos incluem a busca por maior eficiência computacional, facilidade de uso, adição de recursos (como no caso da implementação Extreme Gradient Boosting, que adiciona diferentes métodos de regularização) e até mesmo maior flexibilidade.

Além da definição anterior de Gradient Boosting aplicado a árvores de decisão, este trabalho também utilizou outras implementações de Gradient Boosting, como o Light Gradient Boosting Machine e o Extreme Gradient Boosting. Essas variações partem da mesma premissa do Gradient Boosting, mas adicionam funcionalidades adicionais e são mais eficientes computacionalmente. Assim, esta seção irá definir as diferentes implementações utilizadas e destacar suas principais características.

5.2.6.1 Extreme Gradient Boosting

Extreme Gradient Boosting (CHEN; GUESTRIN, 2016), também conhecido como XGBoost, é um sistema escalável de aprendizagem de máquinas para boosting de árvores de decisão. O sistema está disponível como uma biblioteca de código aberto para diversas linguagens de programação, como C, C++, Python, R, Ruby, JVM, Julia e até mesmo Swift. Aqui foi utilizado a implementação de Python, disponibilizada pela biblioteca `xgboost`.

5.2.6.2 Light Gradient Boosting

6 Metodologia

6.1 Obtenção dos dados

Os dados foram obtidos por meio de web scraping, uma técnica automatizada de extração de informações de páginas web. Para isso, foram utilizadas as linguagens de programação R e Python. No R, os pacotes `xml2` (WICKHAM; HESTER; OOMS, 2023) e `rviz` (WICKHAM, 2024) foram utilizados para extrair dados de páginas estáticas de forma estruturada. No Python, as bibliotecas Scrapy (KOUZIS-LOUKAS, 2016) e Playwright, desenvolvida pela Microsoft, foram empregadas, sendo esta última essencial para a interação com páginas dinâmicas, possibilitando a extração de informações que exigem interações como cliques ou rolagem de página. Além dessas ferramentas, foram implementadas técnicas de rotacionamento de IPs e de modificação das informações do usuário que acessa o site, a fim de evitar bloqueios durante o processo de coleta de dados, garantindo assim a continuidade e eficácia da extração. A ferramenta utilizada para a rotação das informações do usuário que acessa o site é a API criada pela empresa ScrapeOps.

Assim, utilizando as ferramentas e técnicas de web scraping, foram coletadas as variáveis que faziam sentido para a modelagem, priorizando aquelas com menor probabilidade de gerar problemas durante o tratamento dos dados. Ao todo, foram extraídas 25 variáveis, das quais 10 são quantitativas e 15 qualitativas nominais, sendo 13 de caráter dicotômico. No entanto, nem todas as variáveis foram obtidas diretamente por web scraping. As coordenadas de latitude e longitude, por exemplo, foram geradas por meio da geocodificação dos endereços, utilizando o pacote `tidygeocoder` (CAMBON *et al.*, 2021) da linguagem R. Dessa forma, tem-se as seguintes variáveis:

- Valor do imóvel: variável dependente que será modelada e constitui o principal foco de análise deste trabalho;
- Valor médio do aluguel no bairro: valor médio do aluguel dos imóveis no bairro, em m^3 ;
- Área: área total do imóvel, medida em m^2 ;
- Área média do aluguel no bairro: área média dos imóveis alugados no bairro, em m^2 ;

- Condomínio: valor mensal pago pelo condomínio do imóvel;
- IPTU: imposto cobrado sobre imóveis urbanos;
- Banheiros: quantidade de banheiros disponíveis na propriedade;
- Vagas de estacionamento: número total de vagas de estacionamento disponíveis;
- Quartos: quantidade de quartos no imóvel;
- Latitude: posição horizontal, medida em frações decimais de graus;
- Longitude: posição vertical, também medida em frações decimais de graus, assim como a latitude;
- Tipo do imóvel: sete categorias foram consideradas: apartamentos, casas, casas comerciais, casas de condomínio, casas de vila, coberturas, e lotes comerciais e de condomínio;
- Endereço: nome do endereço onde o imóvel está localizado;
- Variáveis dicotômicas: indicam a presença (1) ou ausência (0) de determinadas características no imóvel, como área de serviço, academia, elevador, espaço gourmet, piscina, playground, portaria 24 horas, quadra de esportes, salão de festas, sauna, spa e varanda gourmet.

No entanto, com base nas observações realizadas durante o estudo, nem todas as variáveis coletadas foram utilizadas na modelagem do valor dos imóveis. Algumas foram excluídas devido a uma quantidade excessiva de valores ausentes, enquanto outras se mostraram pouco significativas para explicar o valor do imóvel. Após o processo de coleta e limpeza dos dados, o banco de dados final conta com 31.782 observações.

6.2 Análise exploratória de dados

A análise exploratória de dados é uma das primeiras etapas de qualquer estudo que utiliza a estatística como ferramenta principal, pois permite identificar padrões de comportamento nos dados e descobrir relações entre as variáveis estudadas. Assim, após a coleta e organização dos dados, a primeira etapa deste estudo consistiu em uma análise descritiva. Essa análise possibilitou identificar padrões entre os diferentes tipos de imóveis e como essas características podem influenciar o seu valor. Para evidenciar esses comportamentos, foram criados gráficos e tabelas que permitiram caracterizar as relações entre as variáveis independentes e a variável dependente.

6.3 Construção do modelo

No conjunto de dados extraído, foram avaliados diferentes modelos para a previsão do valor do imóvel. Inicialmente, o valor do imóvel foi explicado por variáveis consideradas relevantes para o estudo, como: valor médio do aluguel no bairro, área, área média do aluguel no bairro, número de banheiros, vagas de estacionamento, número de quartos, latitude, longitude, tipo de imóvel e variáveis dicotômicas obtidas durante o processo de extração. As variáveis relacionadas ao valor do condomínio e IPTU foram excluídas do modelo devido à alta quantidade de valores ausentes.

Com as variáveis selecionadas, o conjunto de dados foi dividido em treino e teste para avaliar o desempenho dos modelos. A divisão foi realizada de forma estratificada, utilizando a classe `StratifiedShuffleSplit`, que garante uma amostragem estratificada e aleatória. A estratificação foi baseada na variável “tipo de imóvel”, preservando a proporção de amostras para cada categoria. Definiu-se 20% do conjunto de dados para o teste, enquanto os 80% restantes foram reservados para o treinamento.

Para a aplicação das ferramentas de modelagem, foram utilizadas as bibliotecas `scikit-learn`, `lightgbm` e `xgboost`. As duas últimas foram empregadas especificamente na modelagem, enquanto a primeira também foi utilizada para criar pipelines de pré-processamento de dados, que organizam etapas sequenciais de preparação necessárias para o tratamento adequado dos dados. Assim, os quatro modelos aplicados na previsão do valor do imóvel foram: Random Forest, Gradient Boosting, LightGBM e XGBoost. Por fim, foi implementado o algoritmo de Stacking, combinando os modelos previamente construídos para melhorar a performance preditiva. No Stacking, foi utilizado como preditor final o algoritmo de Light Gradient Boosting.

6.3.1 Etapas de pré-processamento

Após toda a organização e limpeza dos dados, foram aplicadas algumas transformações com o objetivo de estabelecer uma relação comprehensível das variáveis que mais interferem no valor do imóvel. Além disso, foi aplicado também o tratamento de valores ausentes presentes no conjunto de dados.

O método utilizado para a imputação de valores ausentes foi o algoritmo k-nearest neighbors (KNN). Esse algoritmo estima os valores ausentes de acordo com a fórmula:

$$\hat{y} = \frac{1}{k} \sum_{x_i \in N_k(x)} y_i$$

onde $N_k(x)$ representa o conjunto de k vizinhos mais próximos de x , ou seja, os pontos x_i no conjunto de dados que estão mais próximos de x . Essa proximidade é

geralmente medida pela distância Euclidiana, que é a métrica padrão utilizada pela classe `KNNImputer` da biblioteca scikit-learn para imputação de valores ausentes. No processo de imputação, foi utilizado um total de 17 vizinhos, definido pelo argumento `n_neighbors` da classe `KNNImputer`.

A transformação utilizada para estabilizar a variância e aproximar a distribuição dos regressores de uma distribuição normal foi a transformação logarítmica. Para lidar com variáveis categóricas, foi utilizada a classe `OneHotEncoder`, que aplica a codificação one hot. Essa técnica transforma as categorias das variáveis em variáveis dicotômicas, criando uma nova coluna para cada categoria. Além disso, o `OrdinalEncoder` foi utilizado para variáveis ordinais, que possuem uma ordem. Dessa forma, o `OneHotEncoder` foi aplicado à variável de tipo de imóvel, transformando cada categoria de imóvel em variáveis dummy. Já o `OrdinalEncoder` foi aplicado à variável de tamanho do imóvel, que foi criada posteriormente para classificar os imóveis em diferentes faixas: pequenos, médios e grandes.

Por fim, as variáveis numéricas foram normalizadas utilizando a classe `StandardScaler`, que padroniza os dados aplicando a seguinte transformação:

$$z = \frac{x - \mu}{\sigma}$$

onde μ é a média da amostra de treinamento e σ é o desvio padrão da amostra de treinamento.

6.3.2 Validação cruzada

A técnica utilizada para validar o modelo, além das métricas, foi a validação cruzada. A validação cruzada serve para estimar um erro de generalização médio da seguinte forma $Err = E [L(Y, \hat{f}(X))]$, em que L é uma função perda e \hat{f} é um estimador. Existem diversas técnicas de validação cruzada, a que foi utilizada nesse trabalho é a validação cruzada K-Fold.

A validação cruzada K-Fold é uma técnica que utiliza parte dos dados para ajustar o modelo e outra parte para testá-lo. Nessa abordagem, os dados são divididos em K folds. Em cada iteração, um desses folds é reservado para testar o modelo, enquanto os $K - 1$ folds restantes são usados para treiná-lo. O modelo é ajustado nos $K - 1$ subconjuntos e avaliado no subconjunto de teste, permitindo estimar o erro de predição. Esse processo é repetido K vezes, alternando o subconjunto de teste em cada rodada, e ao final, os K erros de predição são combinados. O erro de predição estimado pela validação cruzada é dado por:

$$CV(\hat{f}) = \frac{1}{N} \sum_{i=1}^N L(y_i, \hat{f}^{-k(i)}(x_i))$$

onde N é o número total de observações, L é a função de perda, y_i é o valor observado, x_i é a entrada correspondente, e $\hat{f}^{-k(i)}$ é o modelo ajustado sem o i -ésimo subconjunto.

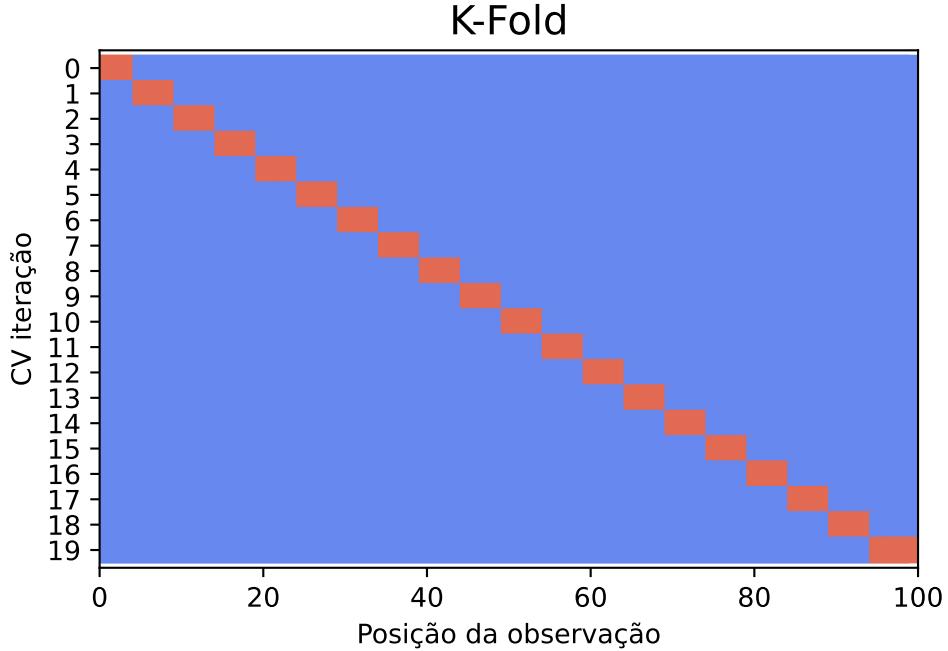


Figura 6.1: Visualização de K-Fold com 20 folds.

A figura acima ilustra exatamente o caso da validação cruzada por K-Fold. As divisões do conjunto de dados para treinamento do modelo são representados pela cor azul, enquanto as laranjas são os conjuntos de validação, onde o modelo ajustado será testado e validado com uma função perda L .

Para aplicar a validação cruzada com K-Fold nos modelos utilizados neste trabalho, foi empregada uma função e uma classe da biblioteca scikit-learn, ambas disponíveis no módulo `model_selection`. A função utilizada para a validação cruzada é a `cross_val_score`, na qual o modelo a ser validado é passado por meio do argumento `estimator`. O critério de avaliação é especificado no argumento `scoring`, e a técnica de validação cruzada é definida pelo argumento `cv`. No caso deste trabalho, foi utilizada a classe `KFold(n_splits=20)`, responsável por realizar a validação cruzada com K-Fold. O argumento `n_splits` define o número de folds utilizados durante a validação, que

neste caso foram 20. Por fim, basta calcular a média das métricas retornadas pelo `cross_val_score` para cada um dos folds.

A métrica utilizada para avaliar o desempenho dos modelos durante a validação cruzada foi a raiz do erro quadrático médio (RMSE). O RMSE mede o quanto os valores estimados pelo modelo se afastam, em média, dos valores observados, sendo que quanto menor o RMSE, melhor é o desempenho do modelo. Além do RMSE, foi também utilizada a métrica MAPE (Erro Percentual Absoluto Médio), que quantifica, em termos percentuais, o desvio médio entre os valores estimados e os observados. Por fim, foi analisado o coeficiente de determinação (R^2), uma medida que indica a proporção da variação da variável dependente explicada pelas variáveis independentes; neste caso, o valor do imóvel. As métricas são definidas da seguinte forma:

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=0}^n (y_i - \hat{y}_i)^2} \quad R^2 = 1 - \frac{SS_{\text{resíduos}}}{SS_{\text{total}}} \quad \text{MAPE} = \frac{1}{n} \sum_{i=0}^n \left| 1 - \frac{y_i}{\hat{y}_i} \right|$$

em que $SS_{\text{resíduos}}$ e SS_{total} representam, respectivamente, a soma dos quadrados dos resíduos e a soma dos quadrados totais.

6.4 Otimização de hiperparâmetros

Existem diversas técnicas para otimização de hiperparâmetros em aprendizado de máquina. Uma das mais comuns é o GridSearch. Segundo BISCHL *et al.* (2023), o GridSearch consiste em dividir o intervalo contínuo de valores possíveis de cada hiperparâmetro em um conjunto de valores discretos, avaliando exaustivamente o algoritmo para todas as combinações possíveis. No entanto, como o número de combinações cresce exponencialmente com o aumento das combinações de hiperparâmetros, o GridSearch apresenta um custo computacional elevado. Por isso, existem métodos de otimização mais sofisticados que oferecem melhor desempenho, como a otimização bayesiana, que foi utilizada neste trabalho.

A otimização bayesiana não se refere a um algoritmo específico, mas sim a uma abordagem de otimização fundamentada na inferência bayesiana, que engloba uma ampla família de algoritmos (GARNETT, 2023). Além disso, a otimização bayesiana tem alcançado benchmarks superiores em comparação com outros algoritmos em diversos problemas complexos de otimização de hiperparâmetros (SNOEK; LAROCHELLE; ADAMS, 2012).

Diferentemente de outros algoritmos de otimização de hiperparâmetros, a otimização bayesiana ajusta suas tentativas futuras de avaliação com base nos resultados obtidos anteriormente (YANG; SHAMI, 2020). Para definir os pontos futuros, utiliza-se uma função probabilística $P(\rho|\lambda)$ (BERGSTRA; YAMINS; COX, 2013). Após ajustar essa função, obtém-se, para cada λ , uma estimativa da performance $\hat{c}(\lambda)$ e da incerteza da predição $\hat{\sigma}(\lambda)$, além da distribuição preditiva associada à função probabilística. Com essa distribuição, uma função de aquisição determina o equilíbrio entre exploitation e exploration¹. Assim, os algoritmos de otimização bayesiana são regidos pela relação $\lambda \rightarrow c(\lambda)$ e buscam um equilíbrio entre exploitation e exploration para identificar as regiões mais promissoras, sem negligenciar possíveis configurações melhores em áreas ainda inexploradas.

6.4.1 Tree-Structured Parzen Estimator

A função probabilística utilizada para a otimização bayesiana neste trabalho foi a Tree-Structured Parzen Estimator (TPE). O TPE define duas funções, $l(x)$ e $g(x)$, que são utilizadas para modelar a distribuição das variáveis do domínio (YANG; SHAMI, 2020). Essas duas densidades são empregadas para estimar a probabilidade de se observar um hiperparâmetro x , dado uma métrica de performance ρ . Assim, tem-se a seguinte definição:

$$p(x|y) = \begin{cases} l(x) & \text{if } y < y^* \\ g(x) & \text{if } y \geq y^* \end{cases} \quad (6.1)$$

em que $l(x)$ representa a densidade quando a função de perda é menor que um limiar y^* , e $g(x)$ é a densidade quando a função de perda tem valores acima de y^* (BERGSTRA *et al.*, 2011). O limite y^* é definido por um hiperparâmetro γ , onde γ corresponde ao percentil dos valores observados de y , de modo que $p(y < y^*) = \gamma$.

Por padrão, o Tree-Structured Parzen Estimator (TPE) utiliza como função de aquisição o Expected Improvement (EI), que pode ser otimizado no TPE da seguinte forma:

$$EI_{y^*}(x) = \int_{-\infty}^{y^*} (y^* - y) p(y|x) dy \quad (6.2)$$

Para encontrar a probabilidade marginal de x , temos a seguinte integral: $p(x) = \int_{\mathbb{R}} p(x|y) p(y) dy$. Particionando o domínio de y , obtemos:

$$p(x) = \gamma l(x) + (1 - \gamma) g(x)$$

¹Exploitation refere-se à busca por soluções próximas a boas observações anteriores, enquanto exploration visa explorar áreas ainda não investigadas

Assim, aplicando o Teorema de Bayes e substituindo na integral da equação Equação 6.2:

$$EI_{y^*}(x) = \int_{-\infty}^{y^*} (y^* - y) \frac{p(x|y)p(y)}{p(x)} dy = \left(\gamma y^* - \int_{-\infty}^{y^*} p(y) dy \right) \left(\gamma + (1 - \gamma) \frac{g(x)}{l(x)} \right)^{-1}$$

A segunda expressão do produto mostra que, para maximizar o Expected Improvement, é necessário encontrar pontos de x com maior probabilidade em $l(x)$ e menor probabilidade em $g(x)$. No entanto, no TPE, maximizar o EI é equivalente a maximizar a razão entre as duas distribuições, definida como $r(x) = \frac{l(x)}{g(x)}$ (COWEN-RIVERS *et al.*, 2022).

6.4.2 Otimização de hiperparâmetros com optuna

Para otimizar os hiperparâmetros dos modelos foi utilizado a bilbioteca `optuna` (AKIBA *et al.*, 2019) da linguagem de programação Python. Essa biblioteca implementa diversos métodos para otimização automatizada de hiperparâmetros. Para a sua utilização, é preciso definir inicialmente uma função objetivo. Por exemplo, para otimizar os hiperparâmetros de uma random forest, seria necessário definir uma função objetivo da seguinte forma:

```
import optuna
import numpy as np
import pandas as pd
from sklearn import ensemble
from sklearn.model_selection import cross_val_score, KFold

def objective(trial):
    X = train_df[vareiables_independentes]
    y = train_df.variavel_dependente

    params = dict(
        n_estimators=trial.suggest_int(
            name='n_estimators',
            low=1,
            high=1000),
        max_depth=trial.suggest_int(
            name='max_depth',
            low=20,
```

```

        high=1000),
        max_features='sqrt',
        random_state=42
    )

model = ensemble.RandomForestRegressor(
    *params
)
model.fit(X=X, y=y)

cv_scores = np.expm1(np.sqrt(-cross_val_score(
    estimator=model,
    X=X,
    y=y,
    scoring="neg_mean_squared_error",
    n_jobs=3,
    cv=KFold(n_splits=20))))

```

return np.mean(cv_scores)

```

study = optuna.create_study()
study.optimize(objective, n_trials=100, n_jobs=-1)

```

Primeiro, define-se, em cada tentativa (trial), quais hiperparâmetros serão otimizados, especificados no objeto `params` no início da função. Após definir o espaço de busca para cada hiperparâmetro, o modelo escolhido é ajustado aos dados de treinamento. Com o modelo ajustado, realiza-se a validação cruzada em cada trial, utilizando o método K-Fold com 20 divisões (folds), conforme definido previamente. Após definir a função objetivo, inicializa-se um estudo com `optuna.create_study` e, em seguida, inicia-se a otimização com `study.optimize(objective, n_trials=100, n_jobs=-1)`. Por fim, para selecionar os melhores hiperparâmetros ao fim do último trial, basta executar `study.best_params`.

Por padrão, a biblioteca Optuna utiliza o Tree-Structured Parzen Estimator (TPE) para otimizar hiperparâmetros de um modelo. A técnica de otimização é escolhida por meio do argumento `sampler` no método `create_study`. Para selecionar o TPE, basta passar `optuna.samplers.TPESampler` como argumento para a criação do estudo. O método TPE é o padrão para otimização na biblioteca optuna. No entanto, caso se deseje utilizar outro método de otimização da biblioteca, basta especificá-lo da mesma forma: `optuna.create_study(sampler=metodo_otimizacao)`.

6.5 Interpretação dos algoritmos de aprendizagem de máquina

Na aplicação de aprendizado de máquina, o foco geralmente está em obter um modelo com o menor erro de generalização possível, o que muitas vezes resulta na negligência da interpretação dos resultados. Isso pode comprometer a compreensão do que o algoritmo está efetivamente fazendo. Em resposta a essa limitação, diversas técnicas têm sido desenvolvidas para interpretar os efeitos das variáveis independentes nas estimativas geradas pelos algoritmos. Assim, esta seção será dedicada a descrever a fundamentação teórica e a aplicação das técnicas de interpretação utilizadas neste trabalho.

6.5.1 Individual Conditional Expectation (ICE)

O método de Individual Conditional Expectation (ICE) é uma ferramenta gráfica utilizada para visualizar as estimativas de um modelo. Esse método traça a relação entre os valores preditos pelo modelo e as variáveis, considerando cada observação individual. Com isso, o ICE permite analisar a variação dos valores ajustados ao longo do intervalo de uma covariável, facilitando a visualização de onde e em que medida pode haver heterogeneidade, além de como cada observação responde individualmente às mudanças em uma variável específica (GOLDSTEIN *et al.*, 2015).

Formalmente, o ICE considera as observações $\{x_{S_i}, \mathbf{x}_{C_i}\}_{i=1}^N$ e os valores preditos \hat{f} . Para cada uma das N observações e valores ajustados x_C , uma curva $\hat{f}_S^{(i)}$ é traçada em função dos valores observados de x_S . Dessa forma, a variável x_S é representada no eixo das abscissas, enquanto os valores observados de \mathbf{x}_C permanecem fixos. Quando há um grande número de linhas no gráfico ICE, a interpretação pode se tornar difícil. Para facilitar a análise, considera-se a centralização das curvas em um ponto específico das variáveis, representando apenas a diferença nas previsões em relação a esse ponto. Essa centralização é conhecida como c-ICE. As novas curvas são então definidas da seguinte forma:

$$\hat{f}_{cent}^{(i)} = \hat{f}^{(i)} - \mathbf{1} \hat{f}(x^*, \mathbf{x}_{C_i})$$

onde x^* é selecionado como o mínimo ou o máximo de x_S , \hat{f} é o modelo ajustado, e $\mathbf{1}$ é um vetor de uns. Quando x^* é o valor mínimo de x_S , isso garante que todas as curvas comecem em 0, removendo assim as diferenças de nível causadas pelos distintos $x_C^{(i)}$. Se x^* for o valor máximo de x_S , o nível de cada curva centralizada reflete o efeito cumulativo de x_S sobre \hat{f} em relação ao ponto de centralização nas variáveis.

6.5.2 Local interpretable model-agnostic explanations (LIME)

RIBEIRO; SINGH; GUESTRIN (2016) definem o Local Interpretable Model-Agnostic Explanations (LIME) como um algoritmo capaz de explicar as previsões feitas por qualquer modelo de classificação ou regressão, aproximando-o localmente de um modelo mais interpretável. Dessa forma, o objetivo geral do LIME é identificar um modelo interpretável e gerar uma representação interpretável para traduzir modelos complexos.

Formalmente, para a construção da explicação produzida pelo LIME, deve-se definir a explicação como um modelo $g \in G$, onde G é uma classe de potenciais modelos interpretáveis, como uma regressão linear ou árvores de decisão. O domínio de g é $\{0, 1\}^{d'}$, isto é, o modelo explicativo g age sobre a ausência ou presença de componentes interpretáveis. Como nem todos os modelos $g \in G$ podem ser simples o suficiente para ser interpretado, define-se uma medida de complexidade $\Omega(g)$ da explicação gerada por $g \in G$. Para uma árvore de regressão, por exemplo, $\Omega(g)$ pode ser a profundidade da árvore, enquanto que para modelos lineares $\Omega(g)$ pode ser o número de pesos diferentes de zero.

Seja $f : \mathbb{R}^d \rightarrow \mathbb{R}$ o modelo que está sendo explicado. Assim, define-se uma medida de aproximação $\pi_x(z)$ entre uma instância z para x , de modo a definir a localidade ao redor de x . Por fim, seja $L(f, g, \pi_x)$ uma estatística do quanto g erra ao tentar aproximar f na localidade definida por π_x . Agora, para garantir a interpretabilidade e a fidelidade local, deve-se minimizar $L(f, g, \pi_x)$, mantendo $\Omega(g)$ suficiente baixo. Assim, tem-se a explicação produzida pelo LIME:

$$\xi(x) = \arg \min_{g \in G} L(f, g, \pi_x) + \Omega(g)$$

Podem existir diversas variações para L e Ω com diferentes famílias de modelos explicativos G . Uma escolha para L , por exemplo, é o erro quadrático médio.

6.5.3 SHapley Additive exPlanations (SHAP)

7 Resultados

7.1 Análise exploratória de dados

A análise exploratória dos dados foi realizada após a divisão entre os conjuntos de treinamento e teste. Essa abordagem foi adotada para evitar o sobreajuste do modelo e garantir que o algoritmo não aprenda com informações indisponíveis no conjunto de teste. Assim, a descritiva dos dados foi realizada utilizando o conjunto de treinamento.

A primeira etapa da análise exploratória de dados foi identificar os dados faltantes e determinar a melhor forma de tratá-los. A Figura 7.1 mostra a porcentagem de observações ausentes em cada variável. As variáveis com a maior quantidade de dados ausentes são o valor do condomínio e o IPTU, pois essas informações são as menos preenchidas no site de onde os dados foram coletados. A terceira variável, com quase 20% de observações ausentes, é a quantidade de vagas de estacionamento. As variáveis com mais de 20% de observações ausentes foram removidas da base de dados, pois, com essa quantidade de valores faltantes, nem mesmo métodos de imputação proporcionariam um tratamento adequado. Dessa forma, apenas as variáveis de valor do condomínio e IPTU foram removidas, enquanto as demais com valores ausentes foram tratadas por meio de imputação.

Uma das dificuldades que podem surgir durante a modelagem é o desbalanceamento das classes, ou seja, a diferença na quantidade de cada tipo de imóvel. O tipo de imóvel mais predominante no conjunto de dados são os apartamentos, que representam 81,36% do total. Em seguida, vêm as casas, com 8,91%, e os flats, com 5,72%. Por fim, as casas comerciais são as menos representadas, com apenas 15 ocorrências. Esse desbalanceamento claro entre as classes pode dificultar o desempenho do modelo, especialmente na previsão de categorias menos frequentes, como as casas comerciais, onde o modelo pode ter dificuldade em obter bons resultados.

A distribuição das variáveis foram analisadas em termos do tipo do imóvel a partir de um gráfico de violino. Pela Figura 7.2, é fácil perceber que a maioria das distribuições possuem assimetria negativa. Os apartamentos possuem caudas longas à direita, indicando presença de valores extremamente altos e que indicam que talvez seja necessário a aplicação de alguma transformação para a estabilização da variância.

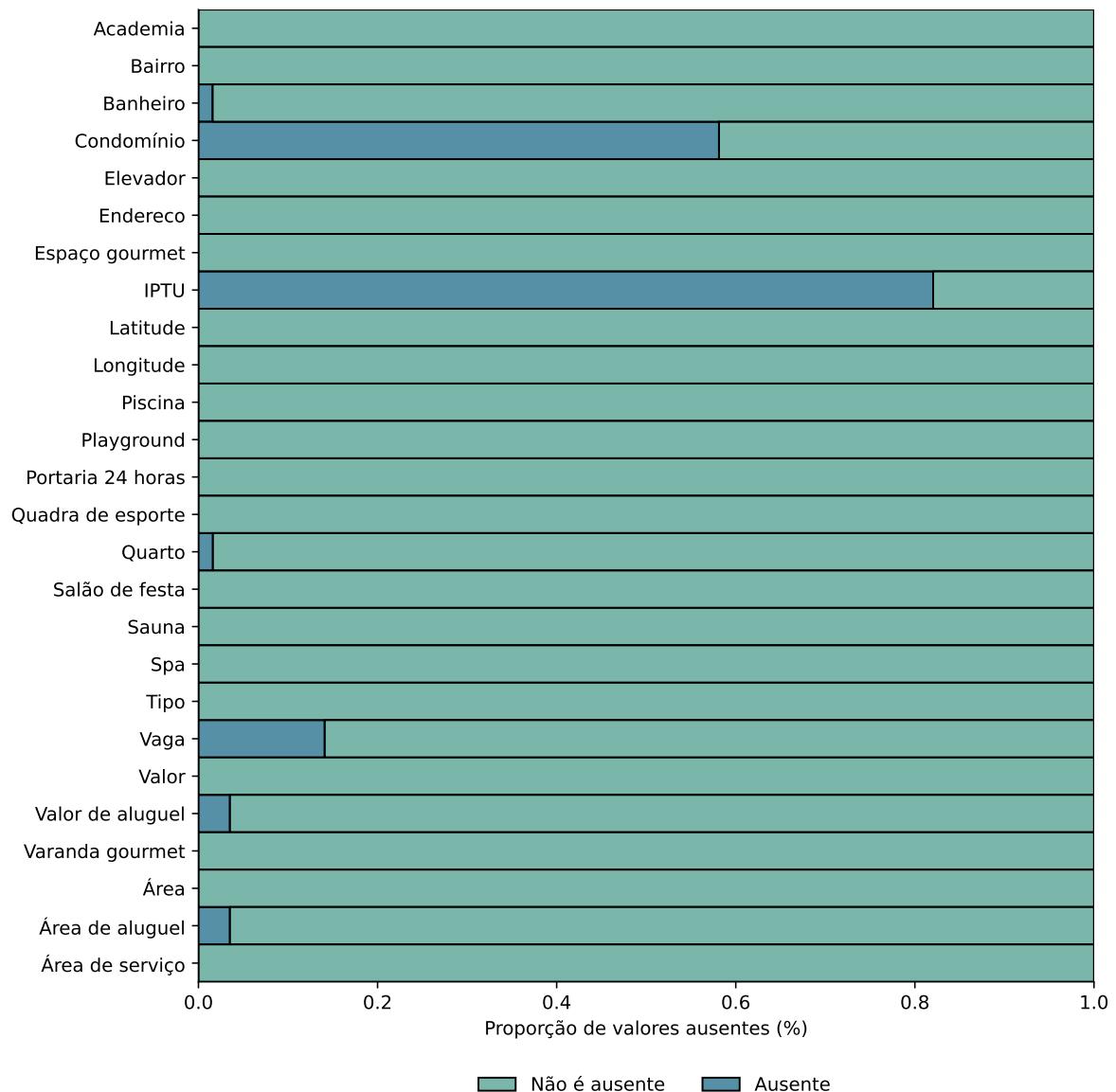


Figura 7.1: Quantidade de valores ausentes por variáveis

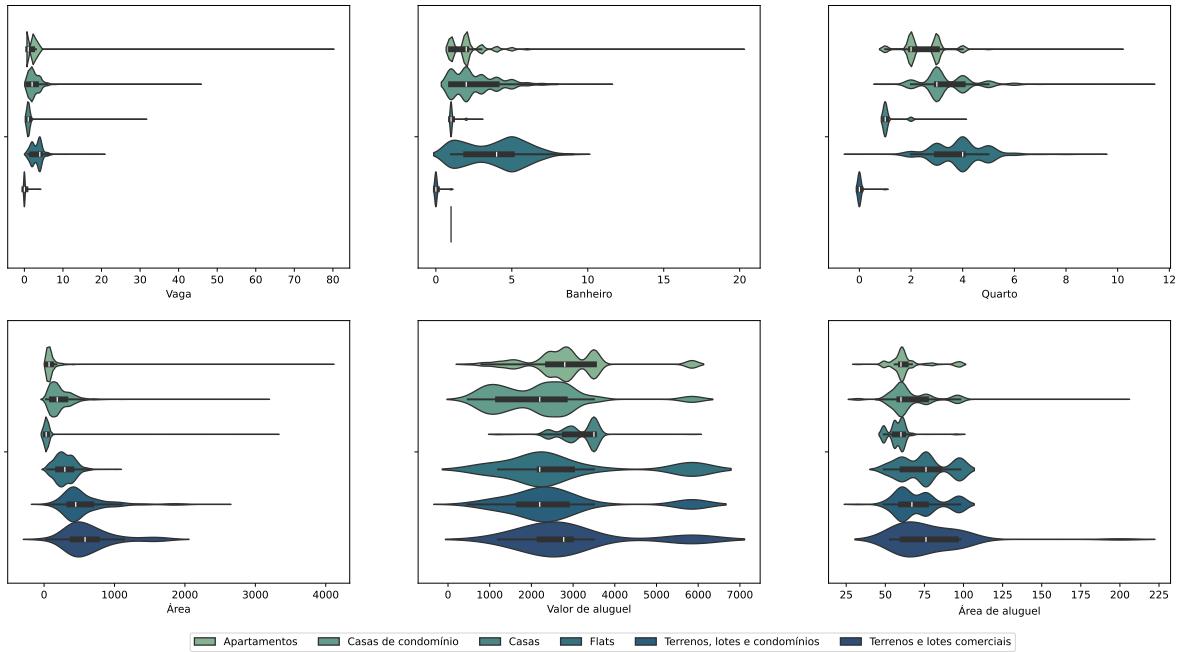


Figura 7.2: Distribuição das variáveis numéricas.

Para reduzir a assimetria da distribuição dos valores dos imóveis, foi aplicada uma transformação logarítmica. O gráfico de densidade à esquerda na Figura 7.3 mostra os dados originais da distribuição do valor dos imóveis. Há uma tendência dos valores ficarem mais concentrados em uma faixa mais baixa, mas alguns imóveis apresentam valores excepcionalmente altos, o que acaba gerando uma distribuição assimétrica positiva. Com a aplicação da transformação logarítmica, a assimetria é suavizada, comprimindo os valores mais altos. Isso tende a aproximar a distribuição de uma forma mais simétrica, facilitando a modelagem e análise estatística.

A Figura 7.4 apresenta a matriz de correlação entre as variáveis numéricas do conjunto de dados. As cores mais escuras indicam uma correlação mais forte entre as variáveis, enquanto as cores mais claras indicam o contrário. O valor do imóvel apresenta maior correlação com as variáveis de área do imóvel e número de vagas de estacionamento. Além disso, o valor do imóvel tem alta correlação com o valor médio do aluguel, número de quartos e banheiros, além de ser fortemente influenciado pela localização das propriedades. Algumas variáveis apresentam multicolinearidade entre si, mas os algoritmos utilizados selecionam aleatoriamente as variáveis para a modelagem, o que reduz o risco de selecionar variáveis redundantes.

ADICIONAR NO FIM UM GRÁFICO DE ESPACIALIZAÇÃO DOS DADOS (INCLUIR VARIAÇÃO DA COR PELO PREÇO DO IMÓVEL)

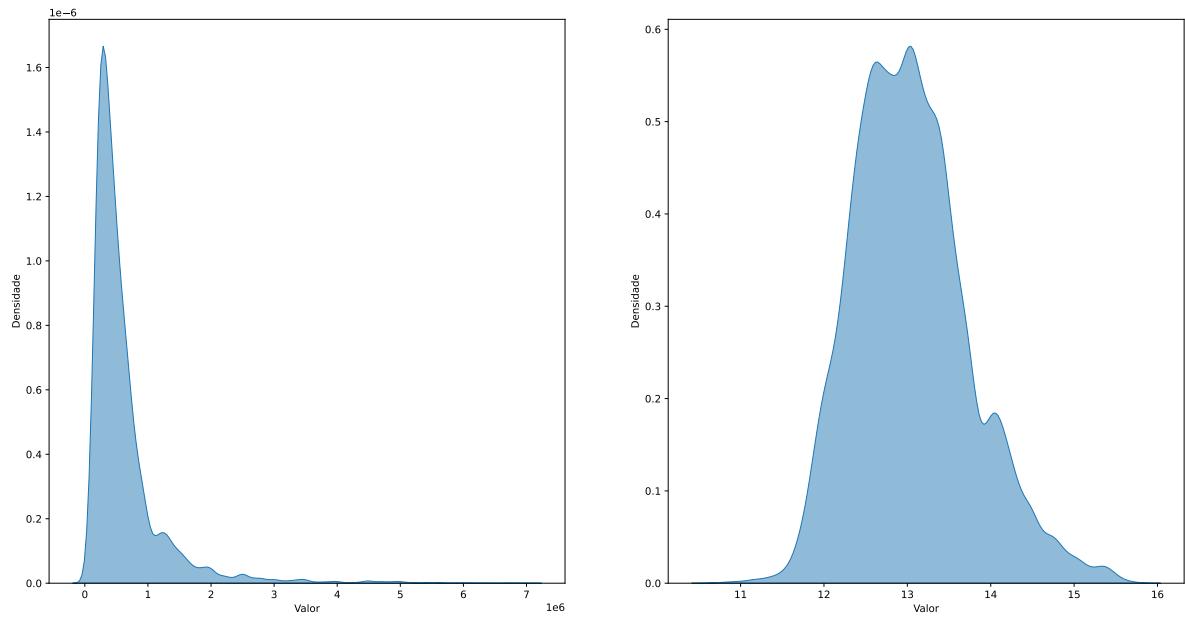


Figura 7.3: Comparação entre distribuição dos valores dos imóveis antes e depois da transformação logarítmica.

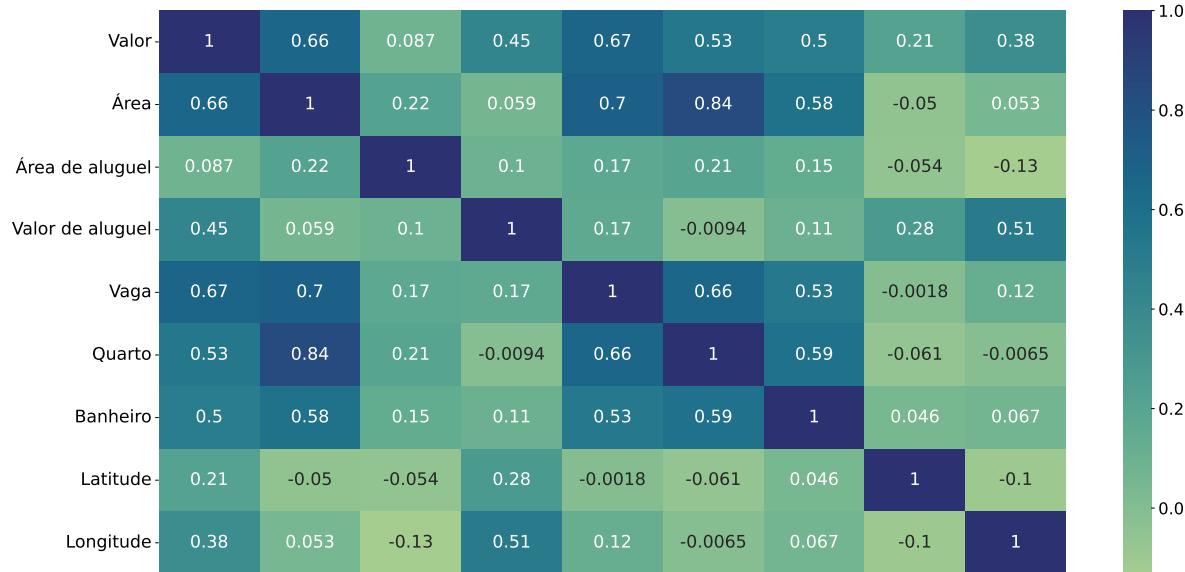


Figura 7.4: Gráfico de correlação de Spearman das variáveis independentes.

7.2 Tunagem dos modelos

Como os algoritmos utilizados neste trabalho são baseados em árvores de decisão ou podem utilizar algoritmos baseados em árvore, como Random Forest, Gradient Boosting e suas variações, os parâmetros escolhidos para otimização serão bastante parecidos. Assim, foram ajustados hiperparâmetros como o número de árvores e a profundidade das árvores, de forma a capturar a complexidade das relações presentes nos dados. Além disso, para os algoritmos baseados em Gradient Boosting, o hiperparâmetro de taxa de aprendizado também foi ajustado.

Todos os modelos e bibliotecas usados neste trabalho seguem a API do scikit-learn. Nessa API, o hiperparâmetro que define a quantidade de árvores é denominado `n_estimators`, o de profundidade das árvores é `max_depth`, e o de taxa de aprendizado é `learning_rate`. Com a configuração da função objetivo na biblioteca Optuna, foi possível encontrar o ponto ótimo desses hiperparâmetros para cada modelo. As métricas obtidas para cada algoritmo pode ser visualizado na Tabela 7.1.

Tabela 7.1: Métricas obtidas de cada algoritmo

Algoritmo	RMSE	R ²	MAPE
Random Forest	0,28972	86,78792%	0,01373
Gradient Boosting	0,28730	86,98259%	0,01377
Light Gradient Boosting	0,28493	87,17132%	0,01346
Extreme Gradient Boosting	0,28659	87,03891%	0,01340
Stacking	0,28473	87,18793%	0,01357

7.2.1 Tunagem da Random Forest

Para o modelo Random Forest, a função objetivo foi definido apenas para otimizar os hiperparâmetros de quantidade de árvores e de profundidade da árvore. O espaço de procura para a quantidade de árvores foi definido entre 1 a 1000 árvores. Já o hiperparâmetro de profundidade da árvore foi definido entre 20 a 1000. Além disso, foi utilizado aleatoriamente $m = \sqrt{p}$ das p variáveis independentes como candidatas para a divisão.

A figura Figura 7.5a mostra a variação da estatística de erro em função dos valores de cada hiperparâmetro ao longo dos trials. O gráfico Figura 7.5b exibe a importância de cada hiperparâmetro, calculada pelo método FANOVA. Na figura Figura 7.5c, observa-se a variação do erro para cada trial, com a linha vermelha representando o menor valor obtido em cada etapa. Por fim, a figura Figura 7.5d é um gráfico de área que mostra a interação entre os hiperparâmetros tunados em relação à estatística de erro.

A partir de Figura 7.5a, é possível ver que o erro tende a ser menor para valores menores do hiperparâmetro de profundidade das árvores (max_depth). Em contraste, para o hiperparâmetro de número de árvores (n_estimators), o erro apresenta uma tendência de redução e estabilização à medida que o número de árvores aumenta. Esse comportamento também é observado em Figura 7.5d, onde valores menores para max_depth e maiores para n_estimators resultam em um modelo com menor erro de generalização.

Além disso, o gráfico de importância dos hiperparâmetros em Figura 7.5b revela que o hiperparâmetro n_estimators possui uma maior contribuição na performance do modelo. A figura Figura 7.5c ilustra o comportamento da métrica de erro para cada trial, mostrando que o método de otimização alcança um de seus melhores valores pouco antes do trial 20. Após esse ponto, o otimizador não consegue encontrar valores de erro significativamente menores.

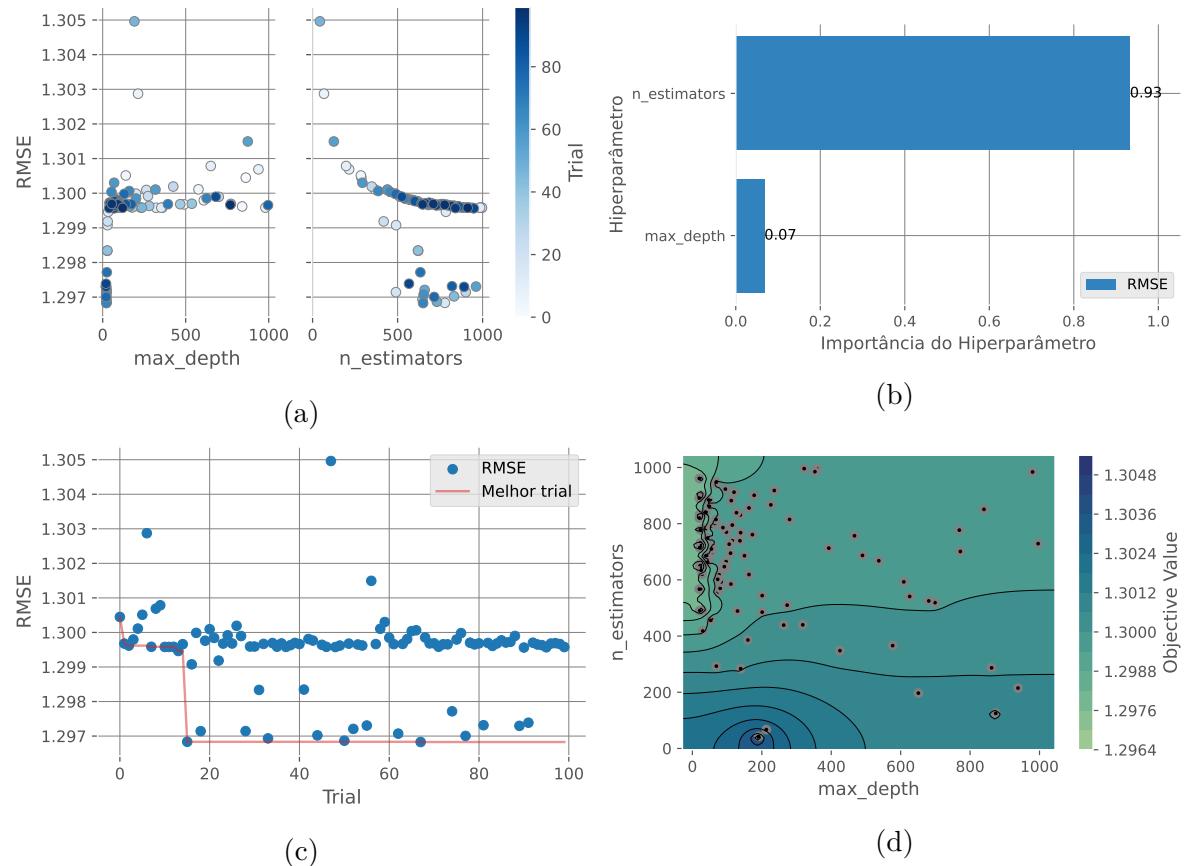


Figura 7.5: Resultados da tunagem da Random Forest.

Tabela 7.2: Melhores hiperparâmetros para Random Forest

Tentativa	RMSE	n_estimators	max_depth
67	1,29682	650	22

7.2.2 Tunagem do Gradient Boosting

Para o algoritmo de Gradient Boosting, os hiperparâmetros ajustados foram a taxa de aprendizado, a quantidade de árvore e a profundidade das árvores. No Optuna, o espaço de busca definido na função objetivo para a taxa de aprendizado variou de $1 \cdot 10^{-5}$ a $1 \cdot 10^{-1}$; para a profundidade das árvores, de 3 a 500; e para a quantidade de árvores, de 50 a 1500. Assim como no Random Forest, foi utilizada $m = \sqrt{p}$ das p variáveis independentes para realizar as divisões nas árvores.

A análise da importância dos hiperparâmetros, apresentada na Figura 7.6b, indica que o hiperparâmetro com maior influência na variação da função objetivo — e, consequentemente, na performance do modelo — é a taxa de aprendizado. Em seguida, a profundidade das árvores é o segundo mais relevante, enquanto o número de árvores tem a menor influência.

Diferentemente do modelo de Random Forest, o algoritmo de Gradient Boosting não apresentou melhorias significativas, como ilustrado na Figura 7.6c, onde a estatística de erro poucas vezes ficou abaixo de 1,3. A relação entre os hiperparâmetros é bastante similar à obtida para o Random Forest. Na Figura 7.6d, observa-se que o método de otimização TPE tende a selecionar valores menores para a profundidade das árvores e maiores para o número de árvores. No entanto, também há uma preferência por uma menor quantidade de árvores quando a taxa de aprendizado diminui.

Tabela 7.3: Melhores hiperparâmetros para Gradient Boosting

Tentativa	RMSE	n_estimators	max_depth	learning_rate
50	1,28891	1500	6	0,08730

7.2.3 Tunagem do LGBM

Para a otimização do LGBM, foram considerados os mesmos hiperparâmetros do algoritmo de Gradient Boosting, mas agora também com a tunagem do número de folhas das árvores. O espaço de busca para a quantidade de árvores foi definido entre 100 e 2000, para a taxa de aprendizado no mesmo intervalo usado no Gradient Boosting, para a profundidade das árvores e número de folhas o intervalo foi definido da mesma forma, entre 100 e 500

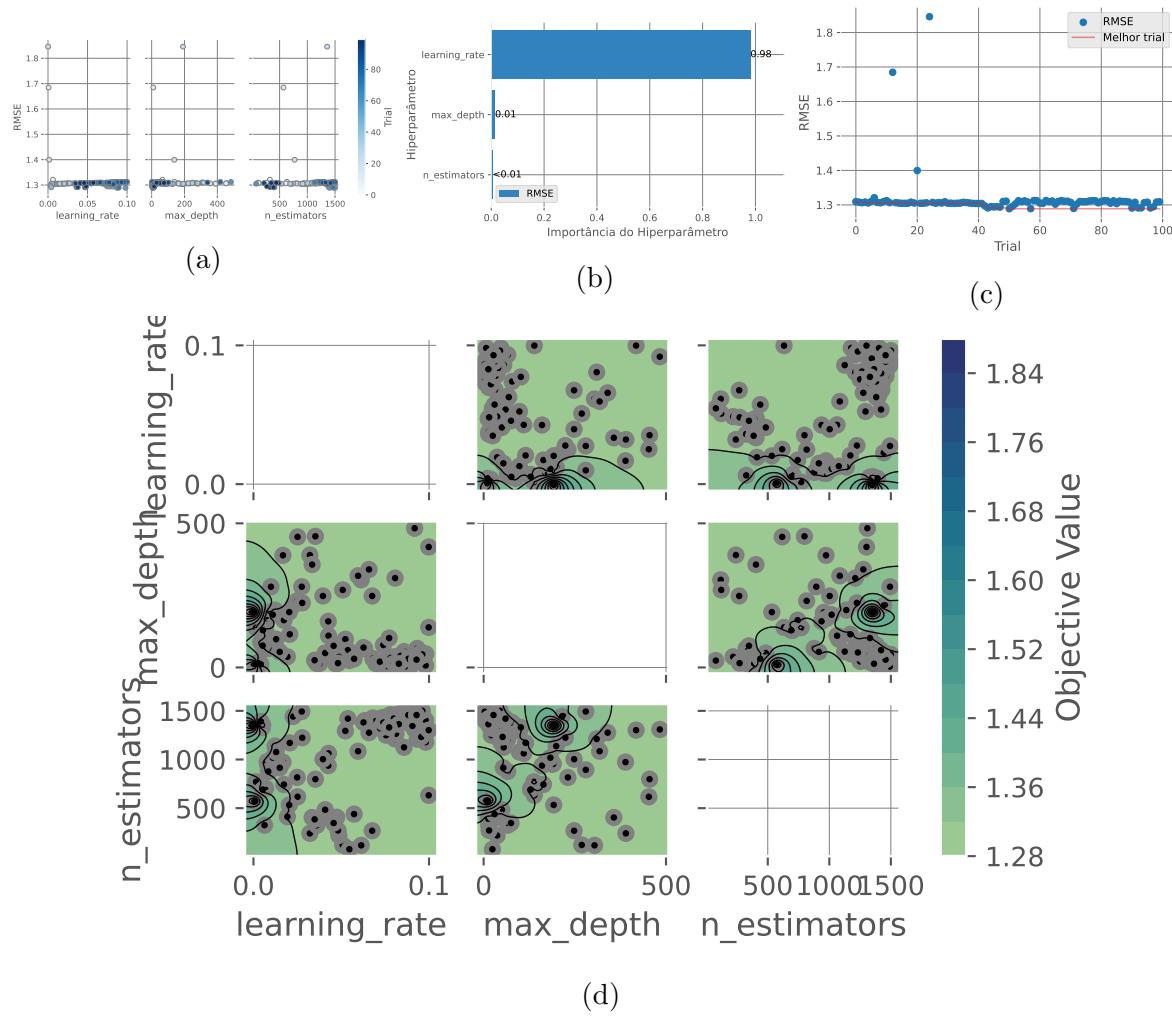


Figura 7.6: Resultados da tunagem do Gradient Boosting

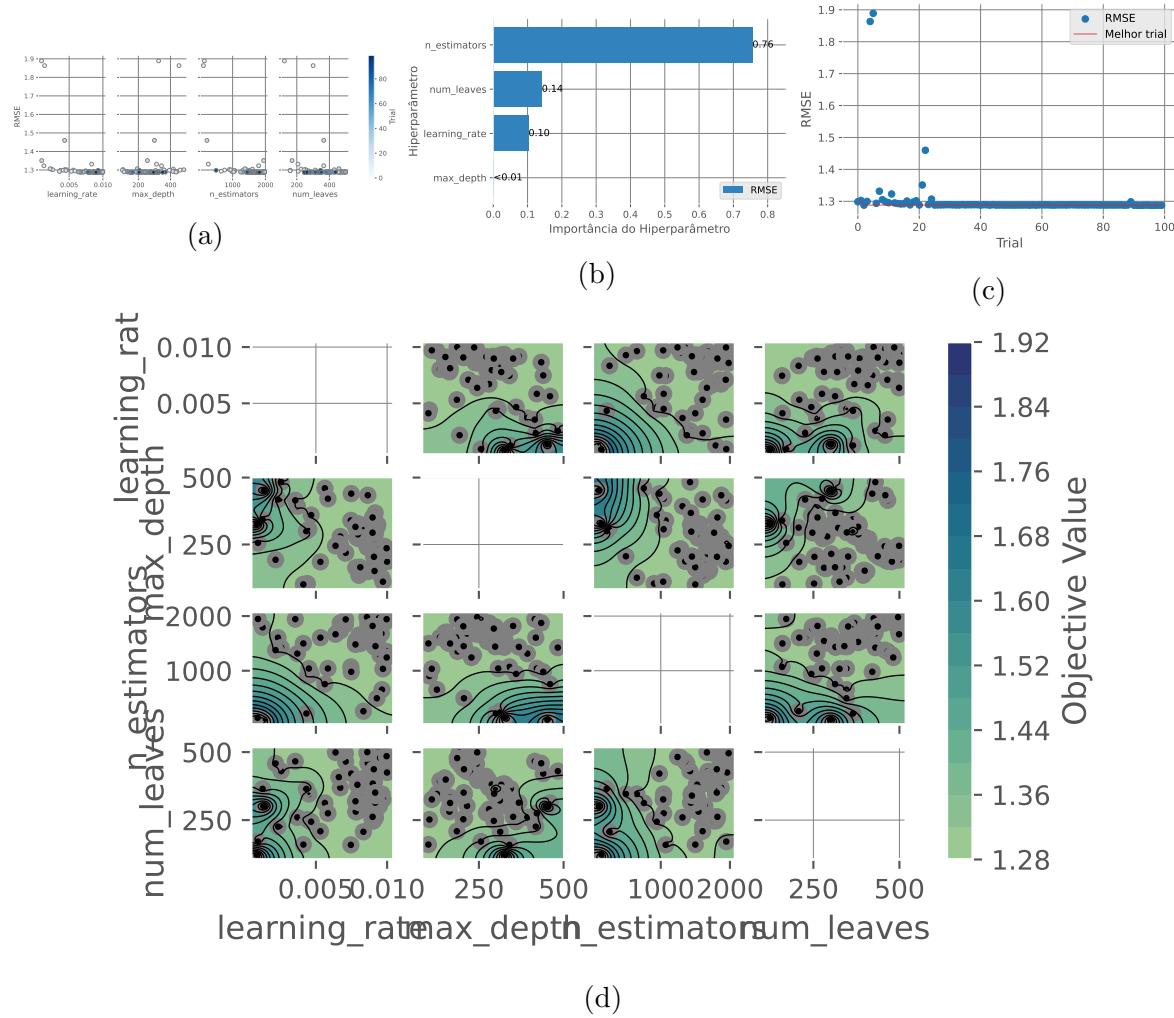


Figura 7.7: Resultados da tunagem do LGBM

As trials de otimização do modelo LGBM (Figura 7.7c) apresentaram um comportamento semelhante ao do Gradient Boosting, porém com maior estabilidade em cada tentativa e uma convergência mais rápida. Ao analisar a contribuição de cada hiperparâmetro para o desempenho do modelo na Figura 7.7b, observa-se que a quantidade de árvores é o hiperparâmetro de maior influência, seguido pela taxa de aprendizado, número de folhas e, por último, a profundidade da árvore.

Analizando a relação entre o hiperparâmetro de profundidade e o número de folhas em função da estatística de erro na Figura 7.7d, observa-se que uma maior quantidade de folhas e uma menor profundidade das árvores resultam em um modelo com erro menor. Esse mesmo comportamento é observado para os outros hiperparâmetros, uma quantidade maior de folhas combinada com uma taxa de aprendizado crescente também produz um modelo com menor erro, assim como um número maior de árvores.

Tabela 7.4: Melhores hiperparâmetros para Light Gradient Boosting

Tentativa	RMSE	n_estimators	num_leaves	max_depth	learning_rate
96	1,28722	1798	247	299	0,00903

7.2.4 Tunagem do XGBoost

Para o algoritmo Extreme Gradient Boosting, os hiperparâmetros selecionados para a tunagem foi a taxa de aprendizagem, profundidade da árvore e quantidade máxima de árvores. Na configuração da função objetivo para a tunagem dos hiperparâmetros pelo optuna, a taxa de aprendizagem variou entre $1 \cdot 10^{-7}$ e 0.5, a profundidade da árvore variou entre 3 e 50 e a quantidade de árvores variou entre 50 e 1000.

A partir da Figura 7.8b, o hiperparâmetro que representou a maior variação na função objetivo, e portanto a maior importância para a performance do modelo, foi a taxa de aprendizagem. A profundidade da árvore representou 22% da variação e por último vem a quantidade de árvores. Observando a Figura 7.8c, é possível ver que o algoritmo de XGBoost foi aquele que apresentou a menor variação entre os trials, com um único trial com erro acima de 2,0.

O algoritmo de otimização TPE apresentou uma maior frequência de procura do hiperparâmetro de taxa de aprendizagem para valores menores, como pode ser visto na Figura 7.8a. O mesmo acontece para o hiperparâmetro de profundidade de árvores, há uma repetição maior para valores menores, indicando que a estatística de erro tendeu a ter valores menores para essa região. Para a quantidade de árvores esse padrão foi diferente, uma quantidade maior de árvores faz com que o modelo tenha um melhor ajuste. Essas relações também podem ser observadas a partir da Figura 7.8d,

menores taxas de aprendizado, combinado com uma menor profundidade de árvore e maior quantidade de árvores apresentam menor valor na função objetivo.

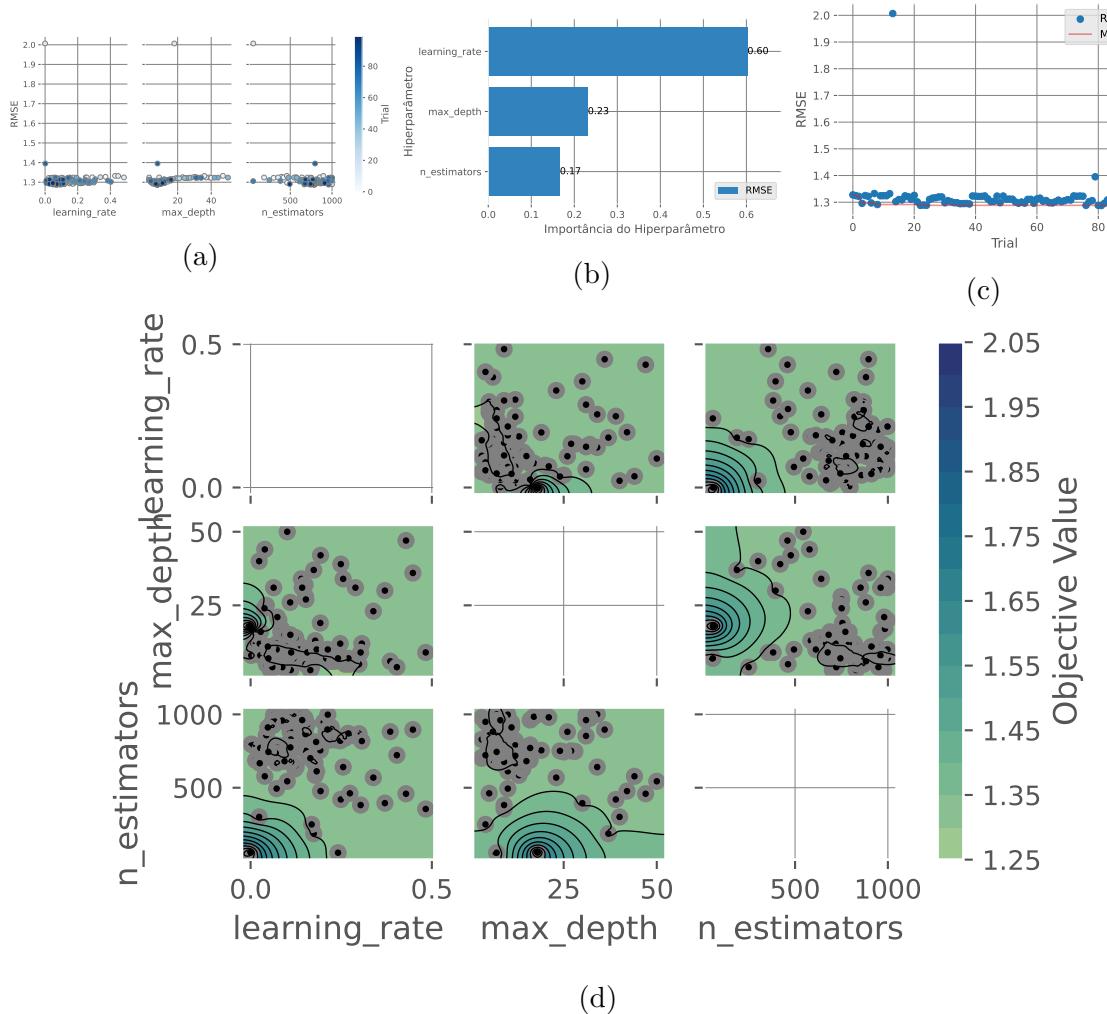


Figura 7.8: Resultados da tunagem do XGBoost

Tabela 7.5: Melhores hiperparâmetros para Extreme Gradient Boosting

Tentativa	RMSE	n_estimators	max_depth	learning_rate
81	1,28752	788	8	0,07119

A tabela Tabela 7.5 apresenta a tentativa em que a combinação de hiperparâmetros resultou no menor erro para o algoritmo de Extreme Gradient Boosting. Nesse caso, a

melhor tentativa foi a de número 81, com uma configuração de 788 árvores, profundidade máxima de 8 e taxa de aprendizado de 0,07119.

Para o LGBM, a melhor combinação foi encontrada na tentativa 96, com 1.798 árvores, 247 folhas, profundidade máxima de 299 e taxa de aprendizado de 0,00903, como apresentado na tabela Tabela 7.4.

Por fim, para os algoritmos Gradient Boosting e Random Forest, as melhores tentativas foram as de números 50 e 67, respectivamente. No caso do Gradient Boosting, a configuração ideal foi composta por 1.500 árvores, profundidade máxima de 6 e taxa de aprendizado de 0,08730. Já para o Random Forest, a melhor combinação consistiu em 650 árvores e profundidade máxima de 22. Os resultados estão na Tabela 7.3 e Tabela 7.2, respectivamente.

7.3 Resultados dos modelos

Nesta seção, serão apresentados os resultados do ajuste de cada algoritmo utilizado: Random Forest, Gradient Boosting, LightGBM e Extreme Gradient Boosting. A análise do ajuste foi realizada com base no gráfico que compara os valores estimados aos valores observados dos imóveis. Além disso, foram utilizadas métricas de erro, como MAPE, R^2 e RMSE, para avaliar o desempenho dos modelos.

O ajuste do algoritmo Random Forest é apresentado na figura Figura 7.9, que exibe a comparação entre os valores previstos e observados. Em termos da transformação logarítmica realizada, a raiz do erro quadrático médio (RMSE) foi de 0,28972, enquanto o coeficiente de determinação R^2 alcançou 86,79, indicando que o modelo explica 86,79 da variação dos dados. Em relação ao MAPE, o valor foi de 0,01373, assim, em média, as previsões realizadas pelo modelo Random Forest, considerando a transformação logarítmica, estiveram 1,373% distantes dos valores reais.

Em relação ao algoritmo de Gradient Boosting, o seu ajuste consegue explicar 86,98% dos dados, bastante semelhante à Random Forest, com uma diferença não tão significativa. Por outro lado, o RMSE do Gradient Boosting foi menor que a Random Forest, tendo sido de 0,28730. Observando o MAPE, não houve também muitas diferenças, embora o MAPE do Gradient Boosting tenha sido maior. O MAPE indica que as previsões geradas pelo Gradient Boosting estão 1,377% distantes de seus valores observados. Seu ajuste pode ser observado na Figura 7.10.

O Light Gradient Boosting conseguiu melhorias em relação aos resultados dos últimos dois algoritmos. Em relação ao seu R^2 , o modelo consegue explicar 87,17% dos dados. O seu RMSE ficou em 0,28493 e o MAPE em 0,01346. Dessa forma, as previsões geradas pelo Light Gradient Boosting estão em média 1,346% distantes de seus valores reais. A melhora do ajuste do Light Gradient Boosting fica bastante perceptível observando a

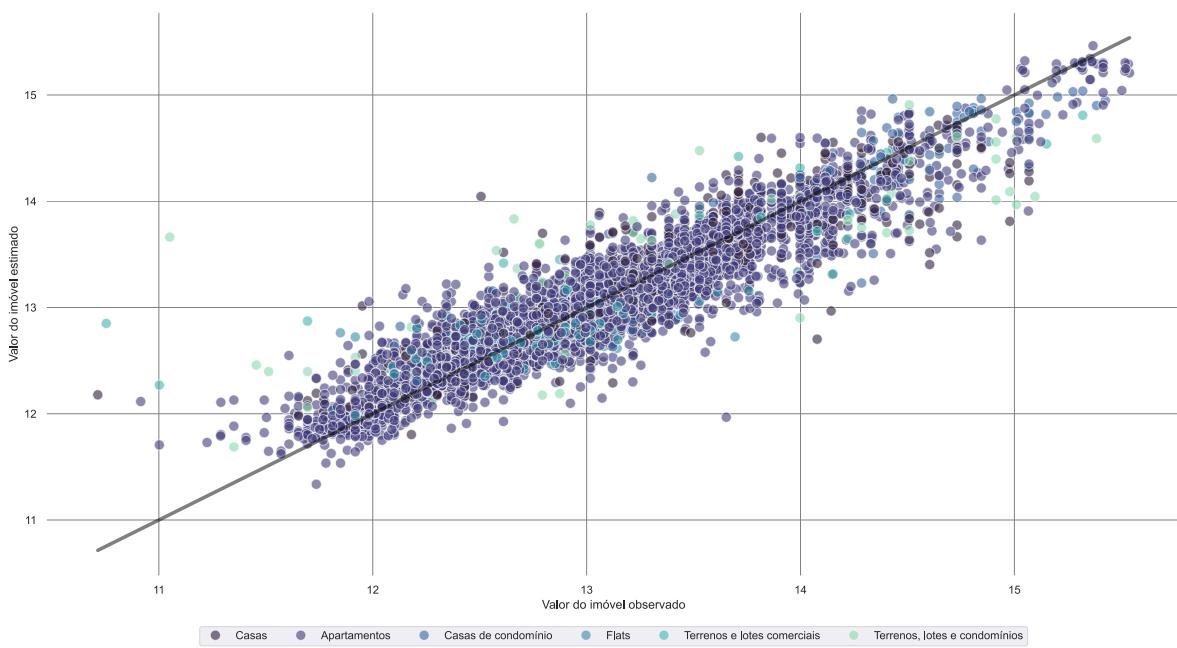


Figura 7.9: Valores previstos em função dos observados do algoritmo Random Forest

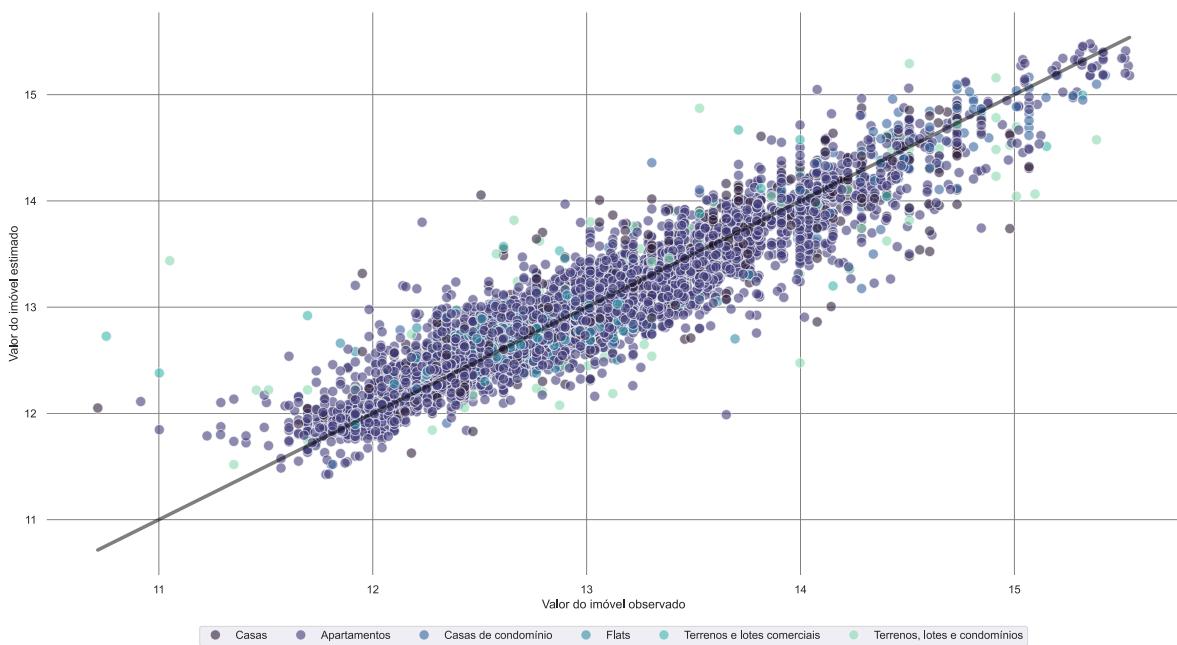


Figura 7.10: Valores previstos em função dos observados do algoritmo Gradient Boosting

figura de seus valores estimados em função dos observados (Figura 7.11). As previsões geradas pelos Gradient Boosting e Random Forest desviam bastante, principalmente em relação ao final da distribuição.

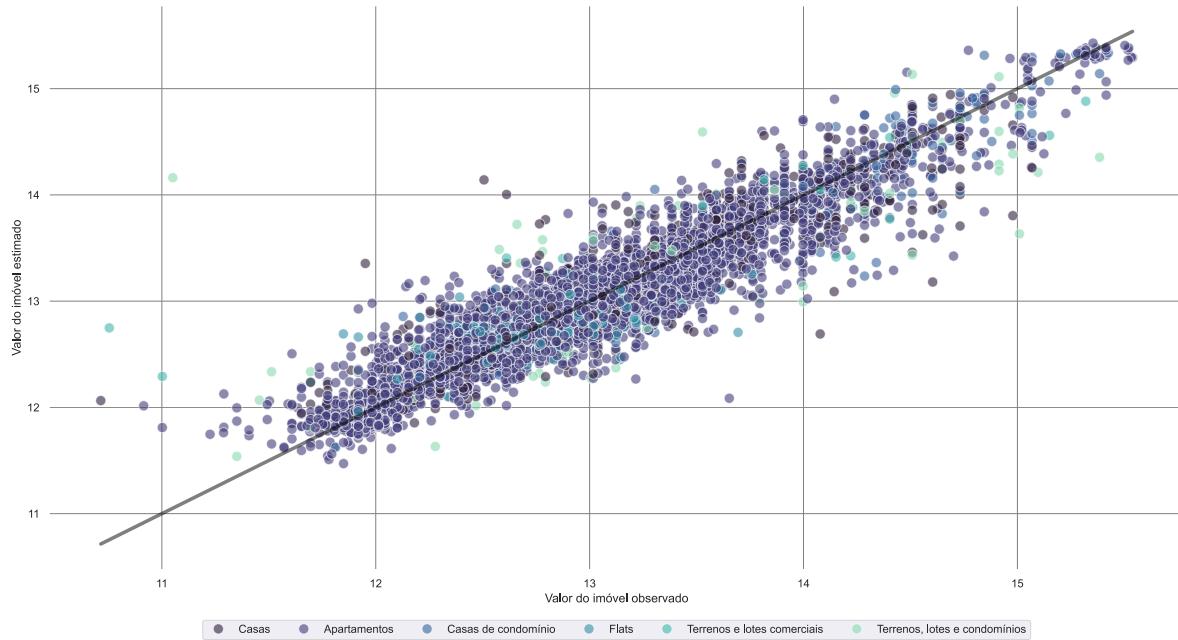


Figura 7.11: Valores previstos em função dos observados do algoritmo Light Gradient Boosting

Assim como o LGBM, o Extreme Gradient Boosting também obteve resultados melhores que os algoritmos de Random Forest e Gradient Boosting. Entretanto, obteve piores em relação às estatísticas de erro de RMSE e R^2 . O RMSE obtido pelo XGBoost foi de 0,28659 e o seu R^2 foi de 87,03891%, indicando que o modelo consegue explicar 87,03891% dos dados. No entanto, obteve um MAPE pouco menor que o LGBM. As previsões geradas pelo Extreme Gradient Boosting estão em média 1,357 distantes de seus valores reais.

Utilizando os últimos modelos com seus hiperparâmetros otimizados e utilizando o Light Gradient Boosting como estimador final, as previsões foram feitas com o Stacking. As previsões geradas pelo Stacking, analisando as métricas de erro utilizadas, foram melhores até mesmo que o Light Gradient Boosting. O RMSE obtido para o Stacking foi de 0,28473 e o modelo consegue explicar 87,18793%. A única métrica que não foi melhor que do algoritmo LGBM foi o MAPE, que foi de 1,357%, mas a diferença é pouca.

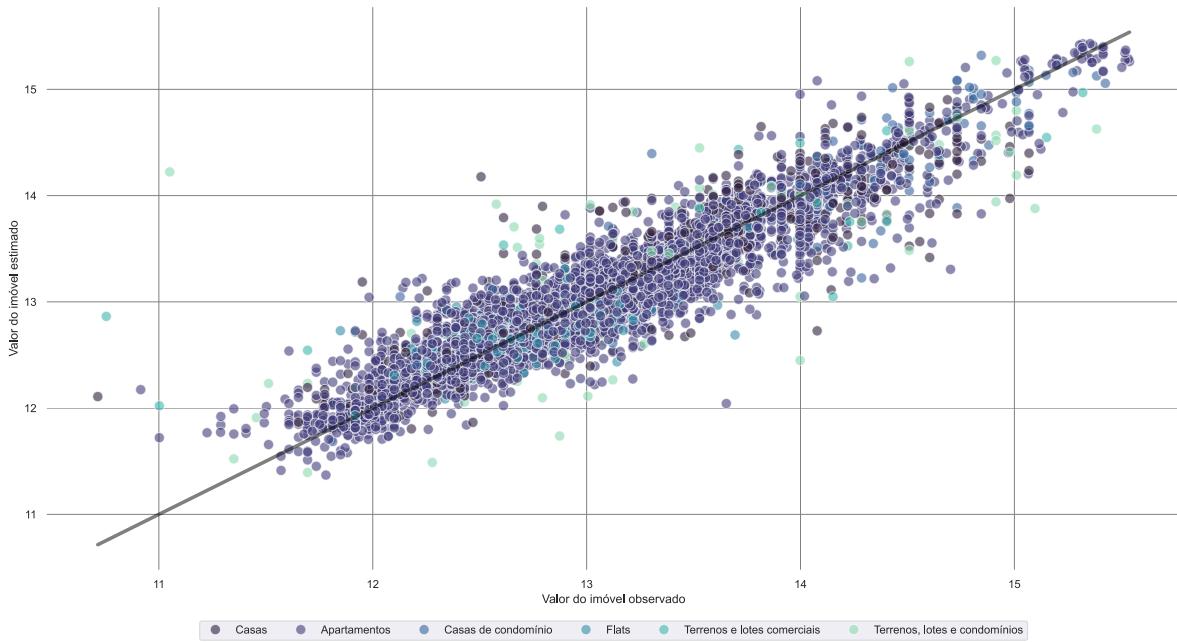


Figura 7.12: Valores previstos em função dos observados do algoritmo Extreme Gradient Boosting

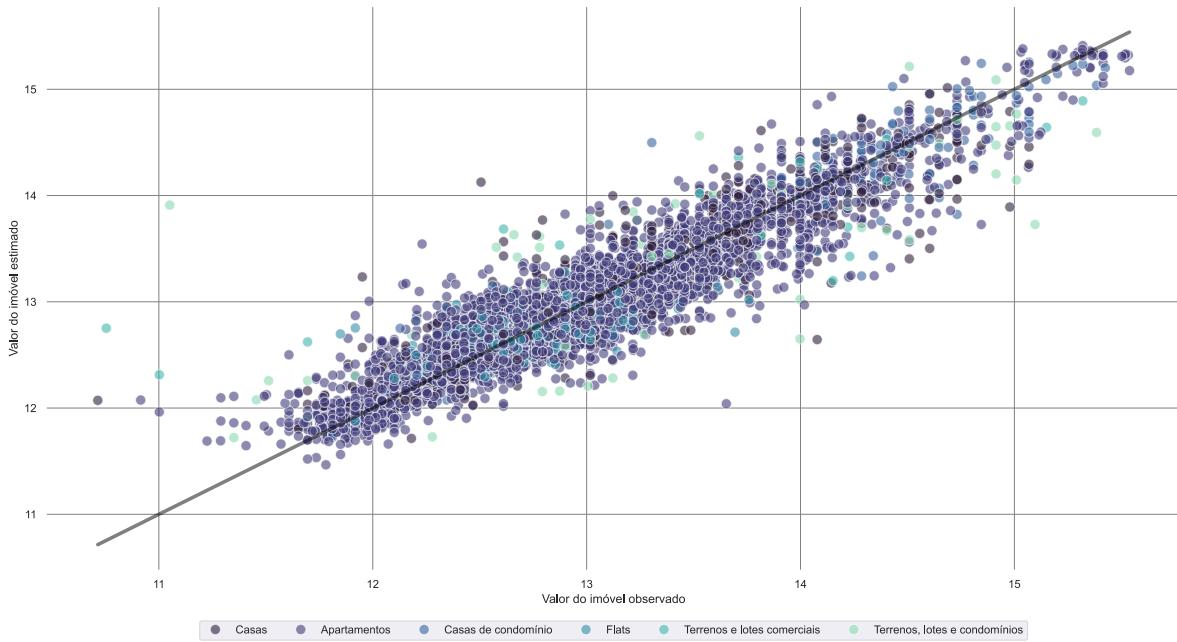


Figura 7.13: Valores previstos em função dos observados do algoritmo Stacking

7.4 Efeito e importância das variáveis na predição

8 Conclusão

9 Referências Bibliográficas

AKIBA, T. *et al.* Optuna: A Next-generation Hyperparameter Optimization Framework. [S.l.]: [s.n.], 2019.

ALLAIRE, J.; DERVIEUX, C. [quarto: R Interface to 'Quarto' Markdown Publishing System](#). [S.l.]: [s.n.], 2024.

BERGSTRA, J. *et al.* Algorithms for hyper-parameter optimization. **Advances in neural information processing systems**, 2011. v. 24.

_____ ; YAMINS, D.; COX, D. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. [S.l.]: PMLR, 2013. p. 115–123.

BISCHL, B. *et al.* Hyperparameter optimization: Foundations, algorithms, best practices, and open challenges. **Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery**, 2023. v. 13, n. 2, p. e1484.

BREIMAN, L. Bagging predictors. **Machine learning**, 1996. v. 24, p. 123–140.

CAMBON, J. *et al.* tidygeocoder: An R package for geocoding. **Journal of Open Source Software**, 2021. v. 6, n. 65, p. 3544. Disponível em: <<https://doi.org/10.21105/joss.03544>>.

CHEN, T.; GUESTRIN, C. Xgboost: A scalable tree boosting system. [S.l.]: [s.n.], 2016. p. 785–794.

COWEN-RIVERS, A. I. *et al.* Hebo: Pushing the limits of sample-efficient hyper-parameter optimisation. **Journal of Artificial Intelligence Research**, 2022. v. 74, p. 1269–1349.

FRIEDMAN, J. H. Stochastic gradient boosting. **Computational statistics & data analysis**, 2002. v. 38, n. 4, p. 367–378.

GARNETT, R. **Bayesian optimization**. [S.l.]: Cambridge University Press, 2023.

GOLDSTEIN, A. *et al.* Peeking inside the black box: Visualizing statistical learning with plots of individual conditional expectation. **Journal of Computational and Graphical Statistics**, 2015. v. 24, n. 1, p. 44–65.

HASTIE, T. *et al.* **The elements of statistical learning: data mining, inference, and prediction.** [S.l.]: Springer, 2009. V. 2.

HUNTER, J. D. **Matplotlib: A 2D graphics environment.** **Computing in Science & Engineering**, 2007. v. 9, n. 3, p. 90–95.

JAMES, G. *et al.* **An introduction to statistical learning.** [S.l.]: Springer, 2013. V. 112.

KOUZIS-LOUKAS, D. **Learning Scrapy.** [S.l.]: Packt Publishing Ltd, 2016.

MCKINNEY, Wes. **Data Structures for Statistical Computing in Python.** (Stéfan van der Walt & Jarrod Millman, Org.). [S.l.]: [s.n.], 2010. p. 56–61.

PEDREGOSA, F. *et al.* Scikit-learn: Machine Learning in Python. **Journal of Machine Learning Research**, 2011. v. 12, p. 2825–2830.

R CORE TEAM. **R: A Language and Environment for Statistical Computing.** Vienna, Austria: R Foundation for Statistical Computing, 2024.

RIBEIRO, M. T.; SINGH, S.; GUESTRIN, C. ”Why should i trust you?” Explaining the predictions of any classifier. [S.l.]: [s.n.], 2016. p. 1135–1144.

RICHARDSON, L. Beautiful soup documentation. **April**, 2007.

SNOEK, J.; LAROCHELLE, H.; ADAMS, R. P. Practical bayesian optimization of machine learning algorithms. **Advances in neural information processing systems**, 2012. v. 25.

TEAM, T. Pandas Development. **pandas-dev/pandas: Pandas.** Zenodo. Disponível em: <<https://doi.org/10.5281/zenodo.3509134>>.

THORPE, A. **selenium: Low-Level Browser Automation Interface.** [S.l.]: [s.n.], 2024.

VAN ROSSUM, G.; DRAKE JR, F. L. **Python reference manual.** [S.l.]: Centrum voor Wiskunde en Informatica Amsterdam, 1995.

WASKOM, M. L. seaborn: statistical data visualization. **Journal of Open Source Software**, 2021. v. 6, n. 60, p. 3021. Disponível em: <<https://doi.org/10.21105/joss.03021>>.

WICKHAM, H. **httr: Tools for Working with URLs and HTTP**. [S.l.]: [s.n.], 2023.

_____. **rvest: Easily Harvest (Scrape) Web Pages**. [S.l.]: [s.n.], 2024.

_____; HESTER, J.; OOMS, J. **xml2: Parse XML**. [S.l.]: [s.n.], 2023.

YANG, L.; SHAMI, A. On hyperparameter optimization of machine learning algorithms: Theory and practice. **Neurocomputing**, 2020. v. 415, p. 295–316.