ESTRUTURA DO PROJETO

CSV:
improved_traffic_data – Dados para a Previsão de Tráfego Urbano
improved_health_data – Dados para a Classificação de Situação de Saúde Pública
improved_sentiment_data – Dados para o Reconhecimento de Emoções
improved_recommendation_data – Dados para o Sistema de Recomendação

Notebooks (ipynb):
urban_traffic_prediction – Notebook da Previsão de Tráfego Urbano
classificacao_saude_publica – Notebook da Classificação de Situação de Saúde Pública
sentiment_analysis – Notebook do Reconhecimento de Emoções
recommendation_system – Notebook do Sistema de Recomendação

Python:
data_gener – Script para melhorar a geração de dados (primeira geração feita pelo Copilot)
mini_dashboard_cidade_inteligente – Dashboard com Streamlit
Terminal:
streamlit run mini_dashboard_cidade_inteligente.py

Joblib:
best_traffic_model – Modelo Tráfego Urbano
best_traffic_model_info - Modelo Tráfego Urbano
traffic_y_test
traffic_y_pred
traffic_prediction_error
best_health_model – Modelo Situação de Saúde Pública
label_encoder_health – Label Encoder Situação de Saúde Pública
best_health_model_info – Parâmetros Situação de Saúde Público
sentiment_model – Reconhecimento de emoções
label_encoder - Reconhecimento de emoções
best_model - Sistema de Recomendação

Códigos:

urban_traffic_prediction – Fiz alterações

* **`sklearn.model_selection`**:

  * `train_test_split`: Divide o conjunto de dados em subconjuntos de treinamento e teste.

  * `RandomizedSearchCV`: Realiza a otimização de hiperparâmetros de forma aleatória.

  * `RepeatedKFold`: Implementa a validação cruzada k-fold repetida, para uma avaliação mais robusta do modelo.

  * `cross_val_score`: Avalia o desempenho de um modelo usando validação cruzada.

* **`sklearn.linear_model.LinearRegression`**: Implementa o modelo de Regressão Linear, um algoritmo base para regressão.

* **`sklearn.ensemble.RandomForestRegressor`**: Implementa o modelo Random Forest para regressão, um algoritmo de ensemble baseado em árvores de decisão.

* **`sklearn.preprocessing.StandardScaler`**: Padroniza as features removendo a média e escalando para a variância unitária.

* **`sklearn.metrics`**: Módulo que oferece diversas métricas para avaliar o desempenho do modelo (mean_squared_error, r2_score).

* **`sklearn.impute.SimpleImputer`**: Preenche valores ausentes em um conjunto de dados.

* **`scipy.stats.zscore`**: Calcula o Z-score de um array, usado para detecção de outliers.

3. Pré-processamento e Limpeza de Dados

* **Imputação de Valores Ausentes (`SimpleImputer`)**:

  * `imputer = SimpleImputer(strategy='most_frequent')`: Inicializa um imputador que preenche valores ausentes com o valor mais frequente (moda) de cada coluna.

  * `data_imputed = pd.DataFrame(imputer.fit_transform(data), columns=data.columns)`: Aplica a imputação aos dados. Esta é uma forma de **tratamento de dados ausentes**.

* **Codificação de Variáveis Categóricas (`pd.get_dummies`)**:

  * `data_encoded = pd.get_dummies(data_imputed, columns=['day_of_week', 'weather', 'event', 'road_condition'], drop_first=True)`: Converte variáveis categóricas (como dia da semana, clima, evento, condição da estrada) em um formato numérico usando **One-Hot Encoding**. `drop_first=True` evita a

multicolinearidade, removendo uma das colunas binárias criadas para cada categoria.

* **Conversão para Numérico (`pd.to_numeric`)**:

  * `data_encoded = data_encoded.apply(pd.to_numeric, errors='coerce')`: Garante que todas as colunas sejam numéricas. `errors='coerce'` converte valores que não podem ser transformados em números para `NaN`, que precisariam de tratamento posterior (embora o código não o faça explicitamente após essa etapa, a imputação já ocorreu).

* **Remoção de Outliers (`zscore`)**:

  * `z_scores = np.abs(zscore(data_encoded[numeric_cols]))`: Calcula o Z-score (número de desvios padrão de distância da média) para cada ponto de dados nas colunas numéricas. O valor absoluto é usado para identificar desvios em ambas as direções.

  * `data_clean = data_encoded[(z_scores < 3).all(axis=1)]`: Filtra as linhas onde **qualquer** Z-score (para **qualquer** coluna numérica) é maior ou igual a 3. Esta é uma técnica de **detecção e remoção de outliers** baseada na regra dos 3 sigmas, que assume uma distribuição aproximadamente normal dos dados.

* **Remoção de Colinearidade**:

  * `corr_matrix = data_clean.corr().abs()`: Calcula a matriz de correlação absoluta entre todas as features.

  * `upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(bool))`: Cria uma máscara para pegar apenas a parte superior triangular da matriz de correlação (para evitar duplicações e a diagonal).

  * `to_drop = [column for column in upper.columns if any(upper[column] > 0.95)]`: Identifica as colunas que têm uma correlação absoluta muito alta (maior que 0.95) com outras colunas.

  * `data_clean = data_clean.drop(columns=to_drop)`: Remove as colunas identificadas para reduzir a **multicolinearidade**. Isso ajuda a evitar que modelos lineares superajustem os dados e melhora a interpretabilidade dos coeficientes.

4. Preparação para Modelagem

* **Separação de Variáveis**:

  * `X = data_clean.drop('travel_time', axis=1)`: Define as features (variáveis independentes).

* `y = data_clean['travel_time']`: Define a variável alvo (variável dependente).

* **Normalização/Escalonamento de Features (`StandardScaler`)**:

* `scaler = StandardScaler()`: Inicializa o padronizador.

* `X_scaled = scaler.fit_transform(X)`: Padroniza as features `X` para ter média zero e desvio padrão um. Isso é crucial para modelos que são sensíveis à escala das features (como Regressão Linear, modelos baseados em distância, e também melhora a convergência de alguns otimizadores).

* **Divisão Treino/Teste (`train_test_split`)**:

* `X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)`: Divide os dados em 80% para treinamento e 20% para teste. `random_state` garante a reprodutibilidade.

5. Treinamento e Otimização de Modelos

* **Validação Cruzada (`RepeatedKFold`)**:

* `rkf = RepeatedKFold(n_splits=5, n_repeats=3, random_state=42)`: Configura a validação cruzada k-fold repetida. Divide os dados em 5 folds (partes) e repete esse processo 3 vezes. Isso fornece uma estimativa mais robusta do desempenho do modelo, reduzindo a variância da estimativa.

* **Modelo 1: Regressão Linear (`LinearRegression`)**:

* `lr_model = LinearRegression()`: Inicializa o modelo de Regressão Linear.

* `lr_scores = cross_val_score(lr_model, X_train, y_train, cv=rkf, scoring='r2')`: Avalia o modelo usando validação cruzada no conjunto de treinamento, medindo o R^2.

* `lr_model.fit(X_train, y_train)`: Treina o modelo final de Regressão Linear com todo o conjunto de treinamento.

* `y_pred_lr = lr_model.predict(X_test)`: Faz previsões no conjunto de teste.

* **Modelo 2: Random Forest Regressor com Otimização de Hiperparâmetros (`RandomizedSearchCV`)**:

* `param_grid = {...}`: Define o espaço de busca para os hiperparâmetros do Random Forest (número de estimadores, profundidade máxima, mínimo de amostras para split e leaf).

* `rf = RandomForestRegressor(random_state=42)`: Inicializa o modelo Random Forest.

* `random_search = RandomizedSearchCV(rf, param_distributions=param_grid, n_iter=50, cv=rkf, scoring='r2', n_jobs=-1)`: Realiza a **busca aleatória de hiperparâmetros**.

* `n_iter=50`: Tenta 50 combinações aleatórias de hiperparâmetros. Mais eficiente que Grid Search para espaços de busca grandes.

* `n_jobs=-1`: Utiliza todos os núcleos da CPU para paralelizar o processo.

* `random_search.fit(X_train, y_train)`: Executa a busca pelos melhores hiperparâmetros no conjunto de treinamento.

* `best_rf = random_search.best_estimator_`: Obtém o modelo Random Forest com os melhores hiperparâmetros encontrados.

* `rf_scores = cross_val_score(best_rf, X_train, y_train, cv=rkf, scoring='r2')`: Avalia o melhor modelo Random Forest usando validação cruzada.

* `y_pred_rf = best_rf.predict(X_test)`: Faz previsões no conjunto de teste.

6. Avaliação do Modelo

* **MSE (Mean Squared Error)**: `mean_squared_error(y_test, y_pred)`: Média dos quadrados dos erros (diferenças entre valores reais e previstos). Penaliza erros maiores. Quanto menor, melhor.

* **R^2 (R-squared)**: `r2_score(y_test, y_pred)`: Coeficiente de determinação. Indica a proporção da variância na variável dependente que é previsível a partir das variáveis independentes. Varia de 0 a 1 (ou pode ser negativo para modelos muito ruins). Quanto mais próximo de 1, melhor o ajuste do modelo.

* Compara o desempenho dos modelos (Regressão Linear e Random Forest) tanto no conjunto de teste quanto com os resultados da validação cruzada.

* Seleciona o "melhor modelo" com base no R^2 no conjunto de teste.

8. Visualização de Resultados

* **Gráfico de Previsão vs. Real (`scatterplot`)**:

* Cria gráficos de dispersão comparando os valores reais (`y_test`) com os valores previstos (`y_pred_lr`, `y_pred_rf`) para ambos os modelos. Um modelo ideal teria todos os pontos na linha y=x.

* **Gráfico de Importância das Variáveis (`barh`)**:

* **`importances = best_rf.feature_importances_`**: Para modelos baseados em árvore como Random Forest, é possível obter a importância de cada feature (o quanto ela contribui para a redução da impureza das folhas).

* Exibe um gráfico de barras horizontais mostrando as features mais importantes para o modelo Random Forest, o que ajuda na **interpretabilidade do modelo**.

classificacao_saude_publica

1. Pré-processamento de Dados

- **Padronização de rótulos** da coluna `risk_level` para letras minúsculas.

- **Codificação de rótulos** com `LabelEncoder` para transformar categorias em valores numéricos.

2. Divisão de Dados

- Separação em **variáveis independentes (X)** e **variável alvo (y)**.

- Divisão em **conjuntos de treino e teste** com `train_test_split`.

3. Balanceamento de Classes

- Aplicação de **SMOTE (Synthetic Minority Over-sampling Technique)** para balancear as classes no conjunto de treino.

4. Normalização

- Uso de `StandardScaler` para padronizar os dados com média 0 e desvio padrão 1.

- O scaler é ajustado apenas no conjunto de treino e aplicado ao teste.

5. Seleção de Atributos

- Uso de `SelectKBest` com `f_classif` para selecionar as melhores features com base em análise univariada.

Modelos testados com `GridSearchCV`:

- **Decision Tree**

- **K-Nearest Neighbors (KNN)**

- **Support Vector Machine (SVM)**

Hiperparâmetros otimizados:

- **Decision Tree**: `max_depth`, `min_samples_split`

- **KNN**: `n_neighbors`, `weights`

- **SVM**: `C`, `kernel`

sentiment_analysis

* **`re`**: Módulo para operações com expressões regulares, essencial para a limpeza de texto.

* **`nltk`**: (Natural Language Toolkit) Biblioteca para processamento de linguagem natural.

* **`unidecode`**: Converte caracteres acentuados ou especiais para sua representação ASCII mais próxima (ex: "ação" para "acao").

* **`nltk.corpus.stopwords`**: Contém uma lista de palavras comuns (stopwords) que são frequentemente removidas do texto.

* **`sklearn.model_selection.train_test_split`**: Divide o conjunto de dados em subconjuntos de treinamento e teste.

* **`sklearn.preprocessing.LabelEncoder`**: Converte rótulos categóricos em rótulos numéricos (0, 1, 2...).

* **`sklearn.pipeline.Pipeline`**: Permite encadear várias etapas de processamento de dados e modelagem em um único objeto, garantindo que as transformações sejam aplicadas consistentemente.

* **`sklearn.feature_extraction.text.TfidfVectorizer`**: Converte uma coleção de documentos brutos em uma matriz de recursos TF-IDF.

* **`sklearn.metrics`**: Módulo que oferece diversas métricas para avaliar o desempenho do modelo (accuracy_score, classification_report, confusion_matrix, f1_score).

* **`sklearn.linear_model.LogisticRegression`**: Implementa o algoritmo de Regressão Logística, um modelo linear para classificação.

* **`imblearn.over_sampling.SMOTE`**: (Synthetic Minority Over-sampling Technique) Técnica de reamostragem para lidar com desequilíbrio de classes.

* **`nltk.download('stopwords')`**: Baixa as palavras irrelevantes (stopwords) para vários idiomas, neste caso, português e inglês.

* **`stop_words = set(stopwords.words('portuguese')).union(set(stopwords.words('english')))`**: Cria um conjunto único de stopwords em português e inglês para remoção eficiente. Usar um `set` melhora a performance da busca.

3. Pré-processamento de Texto (`preprocess` function)

* **`text = str(text)`**: Garante que a entrada seja uma string.

* **`text = unidecode(text)`**: Remove acentos e caracteres especiais, padronizando o texto (ex: "Olá" -> "Ola").

* **`text = re.sub(r'https?://\S+|@\w+|#[\w-]+', '', text)`**: Remove padrões comuns em texto de redes sociais:

  * `https?://\S+`: URLs (links).

  * `@\w+`: Menções de usuários (ex: `@fulano`).

  * `#[\w-]+`: Hashtags (ex: `#exemplo`).

* **`text = re.sub(r'\W|\d', ' ', text.lower())`**:

  * `text.lower()`: Converte todo o texto para minúsculas, garantindo que "Amor" e "amor" sejam tratados como a mesma palavra.

  * `re.sub(r'\W|\d', ' ', ...)`: Remove caracteres não alfanuméricos (`\W`) e dígitos (`\d`), substituindo-os por um espaço. Isso ajuda a isolar as palavras.

* **`tokens = text.split()`**: Divide o texto em uma lista de palavras (tokens).

* **`tokens = [t for t in tokens if t not in stop_words and len(t) > 2]`**: Filtra os tokens:

  * `t not in stop_words`: Remove palavras comuns que não adicionam significado ao sentimento (ex: "o", "a", "de", "e").

  * `len(t) > 2`: Remove palavras muito curtas (geralmente ruído ou caracteres residuais).

* **`return ' '.join(tokens)`**: Junta os tokens limpos de volta em uma única string.

4. Carregamento e Preparação Inicial dos Dados

* **`df.dropna(subset=['text', 'sentiment'], inplace=True)`**: Remove linhas onde as colunas 'text' ou 'sentiment' possuem valores nulos. Isso é uma etapa importante de **limpeza de dados**.

* **`df['text_clean'] = df['text'].apply(preprocess)`**: Aplica a função de pré-processamento `preprocess` a cada texto na coluna 'text', criando uma nova coluna `text_clean` com os textos limpos.

5. Codificação de Rótulos (Label Encoding)

* **`le = LabelEncoder()`**: Inicializa o codificador de rótulos.

* **`df['sentiment_encoded'] = le.fit_transform(df['sentiment'])`**: Converte os rótulos de sentimento categóricos (ex: 'positivo', 'negativo', 'neutro') em valores numéricos inteiros (ex: 0, 1, 2). Isso é necessário porque os algoritmos de Machine Learning geralmente trabalham com entradas numéricas. `fit_transform` aprende os mapeamentos e os aplica.

6. Divisão de Dados (Train-Test Split)

* **`X_train, X_test, y_train, y_test = train_test_split(...)`**: Divide o conjunto de dados em subconjuntos para treinamento e teste.

  * `df['text_clean']`: Os dados de entrada (features) para o modelo.

  * `df['sentiment_encoded']`: Os rótulos de saída (target).

  * `test_size=0.2`: 20% dos dados serão usados para teste, 80% para treinamento.

  * **`stratify=df['sentiment_encoded']`**: Esta é uma técnica crucial para garantir que a proporção de classes no conjunto de treinamento e teste seja a mesma que no conjunto de dados original. Isso é especialmente importante em conjuntos de dados desbalanceados para evitar que um dos conjuntos tenha poucas ou nenhuma amostra de uma classe minoritária.

  * `random_state=42`: Garante a reprodutibilidade da divisão.

7. Vetorização de Texto (TF-IDF)

* **`vectorizer = TfidfVectorizer(...)`**: Transforma o texto limpo em representações numéricas que o modelo pode entender.

* **`ngram_range=(1, 2)`**: Inclui tanto palavras únicas (unigrams) quanto pares de palavras consecutivas (bigrams) como recursos. Bigrams podem capturar mais contexto (ex: "não gosto" é diferente de "gosto").

* **`min_df=2`**: Ignora termos que aparecem em menos de 2 documentos. Isso ajuda a remover palavras muito raras que podem ser ruído ou irrelevantes.

* **`max_df=0.85`**: Ignora termos que aparecem em mais de 85% dos documentos. Isso ajuda a remover palavras muito comuns (mesmo que não sejam stopwords) que podem não ter poder discriminatório (ex: "o" ou "e" se não foram totalmente removidos).

* **`sublinear_tf=True`**: Aplica uma escala logarítmica à frequência de termos (TF), o que significa que o aumento da frequência de uma palavra tem um impacto menor nas pontuações TF-IDF. Ajuda a reduzir o impacto de palavras muito frequentes.

* **`X_train_vec = vectorizer.fit_transform(X_train)`**: `fit_transform` aprende o vocabulário e os pesos TF-IDF do conjunto de treinamento e depois o transforma.

* **`X_test_vec = vectorizer.transform(X_test)`**: `transform` aplica o vocabulário e os pesos TF-IDF *aprendidos no conjunto de treinamento* ao conjunto de teste. É fundamental não usar `fit_transform` no conjunto de teste para evitar vazamento de dados (data leakage).

8. Balanceamento de Classes (SMOTE)

* **`smote = SMOTE(random_state=42)`**: Inicializa o algoritmo SMOTE.

* **`X_train_bal, y_train_bal = smote.fit_resample(X_train_vec, y_train)`**: Aplica SMOTE ao conjunto de treinamento. Esta é uma técnica de **oversampling**, que gera amostras sintéticas da(s) classe(s) minoritária(s) para balancear a distribuição de classes. Isso é crucial para modelos de classificação em dados desbalanceados, pois ajuda o modelo a aprender igualmente sobre todas as classes, evitando o viés em direção à classe majoritária.

9. Treinamento do Modelo (Regressão Logística)

* **`model = LogisticRegression(max_iter=1000)`**: Inicializa o modelo de Regressão Logística.

* **`max_iter=1000`**: Define o número máximo de iterações para o algoritmo de otimização. Um valor maior pode ajudar o modelo a convergir se os dados forem complexos, mas também aumenta o tempo de treinamento.

* **`model.fit(X_train_bal, y_train_bal)`**: Treina o modelo usando os dados de treinamento vetorizados e balanceados.

10. Avaliação do Modelo

* **`y_pred = model.predict(X_test_vec)`**: Realiza previsões no conjunto de teste.

* **`accuracy = accuracy_score(y_test, y_pred)`**: Calcula a **acurácia**, que é a proporção de previsões corretas.

* **`f1 = f1_score(y_test, y_pred, average='weighted')`**: Calcula o **F1-score**.

  * O F1-score é a média harmônica de precisão e recall. É uma métrica mais robusta que a acurácia para conjuntos de dados desbalanceados.

  * `average='weighted'`: Calcula o F1-score para cada classe e depois pondera pela proporção de cada classe no conjunto de dados.

* **`conf_matrix = confusion_matrix(y_test, y_pred)`**: Gera a **matriz de confusão**, que mostra o número de verdadeiros positivos, verdadeiros negativos, falsos positivos e falsos negativos. É fundamental para entender o desempenho do modelo em cada classe.

* **`class_report = classification_report(y_test, y_pred, target_names=le.classes_)`**: Gera um **relatório de classificação** detalhado, incluindo precisão (precision), recall, F1-score e suporte (número de ocorrências) para cada classe. `target_names=le.classes_` exibe os nomes originais dos sentimentos em vez de 0, 1, 2.

12. Visualização (Matriz de Confusão)

* **`sns.heatmap(...)`**: Utiliza a biblioteca `seaborn` para criar um mapa de calor visual da matriz de confusão.

  * `annot=True`: Mostra os valores numéricos em cada célula.

  * `fmt="d"`: Formata os números como inteiros.

  * `cmap="Blues"`: Define o esquema de cores.

  * `xticklabels=le.classes_`, `yticklabels=le.classes_`: Define os rótulos dos eixos com os nomes originais dos sentimentos.

* **`plt.xlabel(...)`, `plt.ylabel(...)`, `plt.title(...)`**: Adiciona rótulos e um título ao gráfico para melhor clareza.

* **`plt.show()`**: Exibe o gráfico.

recommendation_system

* **`sklearn.metrics.pairwise.cosine_similarity`**: Calcula a similaridade do cosseno entre vetores, a métrica central para encontrar usuários semelhantes.

2. Carregamento e Pré-processamento de Dados

* **`data = data[~data['user'].isin(['User1', ..., 'User5'])]`**: **Filtragem de Dados**. Remove usuários específicos (User1 a User5) do conjunto de dados. Isso pode ser feito para remover ruído, bots, ou usuários de teste.

3. Engenharia de Features: Interesse Ponderado

* **`data['weighted_interest'] = data['rating'] * data['frequency']`**: Cria uma nova feature, `weighted_interest` (interesse ponderado), multiplicando a `rating` (avaliação) pela `frequency` (frequência). Esta é uma forma simples de **engenharia de features** que tenta capturar um nível mais robusto de engajamento do usuário com um serviço do que apenas a avaliação bruta.

4. Criação da Matriz Usuário-Serviço

* **`user_service_matrix = data.pivot_table(index='user', columns='service', values='weighted_interest', aggfunc='mean', fill_value=0)`**: Transforma os dados em uma **matriz de interesse usuário-serviço**.

  * `index='user'`: Usuários se tornam os índices das linhas.

  * `columns='service'`: Serviços se tornam as colunas.

  * `values='weighted_interest'`: Os valores dentro da matriz são os interesses ponderados.

  * `aggfunc='mean'`: Se um usuário tiver múltiplos registros para o mesmo serviço, a média do interesse ponderado será usada.

  * `fill_value=0`: Preenche os valores ausentes (onde um usuário não interagiu com um serviço) com 0.

5. Normalização da Matriz

* **`user_service_matrix_normalized = user_service_matrix.div(user_service_matrix.sum(axis=1), axis=0)`**: **Normalização por Linha**. Normaliza cada linha (usuário) da matriz `user_service_matrix`. Isso significa que o interesse total de cada usuário será 1. A normalização ajuda a lidar com o "problema de escala" (alguns usuários avaliam/interagem mais do que outros), tornando as comparações de similaridade mais justas.

6. Cálculo de Similaridade entre Usuários (Similaridade do Cosseno)

* **`user_similarity = cosine_similarity(user_service_matrix)`**: Calcula a **similaridade do cosseno** entre todos os pares de usuários na matriz de interesse não normalizada.

* **`user_similarity_df = pd.DataFrame(...)`**: Converte a matriz de similaridade NumPy em um DataFrame Pandas com rótulos de usuário.

* **`user_similarity_normalized = cosine_similarity(user_service_matrix_normalized)`**: Repete o cálculo da similaridade do cosseno para a matriz de interesse *normalizada*.

* **`user_similarity_normalized_df = pd.DataFrame(...)`**: Converte a matriz de similaridade normalizada em um DataFrame.

* **Similaridade do Cosseno**: Mede o ângulo entre dois vetores. Se os vetores são na mesma direção (usuários com gostos muito semelhantes), o cosseno é 1. Se são opostos, é -1. Se são ortogonais (sem relação), é 0. É uma métrica comum para dados esparsos como os gerados por filtragem colaborativa.

7. Função de Recomendação (`recommend_services`)

Esta função implementa a lógica central da filtragem colaborativa:

* **Identificação de Usuários Semelhantes**: `similarity_df[user].nlargest(top_n + 1).index[1:]` encontra os `top_n` usuários mais semelhantes ao usuário alvo (excluindo o próprio usuário).

* **Cálculo de Scores Ponderados**: `sum(similarity_df[user][sim_user] * matrix.loc[sim_user] for sim_user in similar_users)`: Para cada serviço, ele soma os interesses dos usuários semelhantes, ponderando pelo quão semelhantes eles são ao usuário alvo. Isso é o cerne da filtragem colaborativa.

* **Filtragem de Serviços Já Utilizados**: `user_services[user_services.isnull() | (user_services == 0)].index` identifica os serviços que o usuário alvo ainda não utilizou (ou cujo interesse ponderado é 0). Isso evita recomendar algo que o usuário já conhece.

* **Geração de Recomendações**: `weighted_scores.loc[not_rated].nlargest(top_n)` seleciona os `top_n` serviços com os maiores scores ponderados dentre aqueles que o usuário ainda não utilizou.

* **Fallback para Serviços Populares**:

  * Se o usuário já utilizou todos os serviços, ou

  * Se nenhuma recomendação personalizada puder ser gerada (ex: nenhum serviço não avaliado tem score positivo),

  * O sistema recomenda os `top_n` serviços mais populares (`matrix.sum().sort_values(ascending=False).head(top_n)`). Isso é uma estratégia de **tratamento de cold start para itens** ou **fallback para usuários com poucas interações**.

* **Tratamento de Usuário Desconhecido**: Verifica se o usuário está na matriz e retorna um erro se não estiver, uma forma básica de **tratamento de cold start para usuários**.

8. Métricas de Avaliação (`precision_recall_at_k`)

Esta função calcula a **Precision@K** e **Recall@K**, métricas comuns para avaliar sistemas de recomendação:

* **`recommendations = recommend_services(...)`**: Obtém as recomendações para o usuário.

* **`relevant_services = matrix.loc[user][matrix.loc[user] >= threshold]`**: Define o que é um serviço "relevante" para o usuário. Neste caso, um serviço é relevante se o interesse ponderado do usuário por ele for maior ou igual a um `threshold` (limiar). Isso simula o "verdadeiro interesse" do usuário.

* **`interseção = set(recommended_services).intersection(set(relevant_services.index))`**: Calcula quantos dos serviços recomendados são realmente relevantes.

* **Precision@K**: `len(interseção) / len(recommended_services)`. A proporção de recomendações que são relevantes. Um Precision@K alto significa que o sistema é bom em não recomendar coisas irrelevantes.

* **Recall@K**: `len(interseção) / len(relevant_services)`. A proporção de serviços relevantes que foram encontrados pelo sistema. Um Recall@K alto significa que o sistema é bom em encontrar a maioria das coisas que o usuário acharia relevantes.

* **Tratamento de Casos Vazios**: Retorna 0.0, 0.0 se não houver recomendações ou serviços relevantes.

9. Teste e Exibição de Resultados

* **`for user in users:`**: O sistema é testado para os primeiros 3 usuários.

* Calcula e exibe as recomendações, Precision@K e Recall@K para ambas as abordagens (com e sem normalização), permitindo comparar o impacto da normalização.

10. Visualização de Dados

* **`sns.heatmap(...)`**: Gera mapas de calor para visualizar a matriz de similaridade entre usuários (com e sem normalização). Isso ajuda a entender visualmente como os usuários se agrupam por gostos semelhantes.

* **`sns.barplot(...)`**: Gera um gráfico de barras da popularidade geral dos serviços, mostrando quais serviços são mais utilizados/avaliados no conjunto de dados.

Mini_dashboard_cidade_inteligente.py

1. Importações de Bibliotecas

- **streamlit as st**: A biblioteca principal para construir a aplicação web interativa.
- **sklearn.model_selection**:
    - train_test_split: Para dividir dados em conjuntos de treino e teste.
    - GridSearchCV: Para otimização de hiperparâmetros (não diretamente usado nos módulos apresentados, mas foi usado na fase de treinamento externa).
    - cross_val_score: Para avaliação do modelo usando validação cruzada (também para fase de treinamento externa).

- **sklearn.preprocessing**:
    - LabelEncoder: Para codificar rótulos categóricos em numéricos.
    - StandardScaler: Para padronizar features numéricas.
- **sklearn.tree.DecisionTreeClassifier**: Um tipo de classificador (modelo de ML).
- **sklearn.neighbors.KNeighborsClassifier**: Outro tipo de classificador.
- **sklearn.svm.SVC**: Classificador Support Vector Machine.
- **sklearn.linear_model.LinearRegression**: Modelo de regressão linear.
- **sklearn.ensemble.RandomForestRegressor**: Modelo de regressão Random Forest.
- **sklearn.metrics**:
    - accuracy_score: Mede a acurácia de um classificador.
    - confusion_matrix: Cria a matriz de confusão para classificadores.
    - classification_report: Gera um relatório completo de métricas de classificação.
    - mean_squared_error: Mede o erro quadrático médio para regressão.
- **sklearn.feature_extraction.text.TfidfVectorizer**: Converte texto em representações numéricas TF-IDF.
- **sklearn.pipeline.Pipeline**: Permite encadear múltiplas etapas de processamento e modelagem.
- **imblearn.over_sampling.SMOTE**: Técnica para lidar com desequilíbrio de classes (usada na fase de treinamento do modelo de sentimento).
- **re**: Para operações com Expressões Regulares (usado no pré-processamento de texto).
- **sklearn.metrics.pairwise.cosine_similarity**: Para calcular similaridade entre vetores (usado no sistema de recomendação).
- **sklearn.linear_model.LogisticRegression**: Classificador de Regressão Logística.
- **sklearn.naive_bayes.MultinomialNB**: Classificador Naive Bayes para dados de texto.
- **nltk**: (Natural Language Toolkit) Para processamento de linguagem natural.

- **nltk.corpus.stopwords**: Contém listas de palavras irrelevantes.
- **nltk.stem.RSLPStemmer**: Algoritmo de "stemming" para a língua portuguesa.

---

2. Configuração Inicial e Pré-processamento Compartilhado

- **nltk.download('stopwords')**: Baixa as stopwords do NLTK (se ainda não tiverem sido baixadas).
- **stop_words = set(stopwords.words('portuguese'))**: Carrega as stopwords em português em um set para busca eficiente.
- **stemmer = RSLPStemmer()**: Inicializa o stemmer para português. O *stemming* reduz palavras à sua "raiz" ou radical (ex: "correndo", "corria" -> "corr").
- **Função preprocess(text)**: Esta função é reutilizada em diferentes módulos que lidam com texto.
    - re.sub(r'\W|\d', ' ', text.lower()): Converte o texto para minúsculas, remove caracteres não alfanuméricos (\W) e dígitos (\d), substituindo-os por espaços.
    - tokens = text.split(): Divide o texto em palavras (tokens).
    - tokens = [stemmer.stem(word) for word in tokens if word not in stop_words and len(word) > 2]: Filtra os tokens:
        - Remove stopwords.
        - Remove palavras com 2 ou menos caracteres (geralmente ruído).
        - Aplica o *stemming* a cada palavra restante.
    - return ' '.join(tokens): Junta os tokens limpos de volta em uma string.

---

3. Carregamento de Dados Iniciais

Quatro datasets são carregados no início da aplicação. Cada um é usado por uma seção específica:

- health_data.csv
- traffic_data.csv

- sentiment_data.csv
- recommendation_data.csv
- **recommendation_data = recommendation_data[~recommendation_data['user'].isin(['User1', ..., 'User5'])]**: Filtra alguns usuários de teste/indesejados do dataset de recomendação.

---

4. Estrutura da Aplicação Streamlit (Menu Lateral)

- **st.sidebar.title("Cidade Inteligente - Lumenópolis")**: Define o título na barra lateral da aplicação.
- **app_selection = st.sidebar.selectbox("Selecione a aplicação:", [...])**: Cria um menu selectbox na barra lateral, permitindo ao usuário escolher qual das quatro aplicações (Previsão de Tráfego, Classificação de Saúde, Análise de Sentimentos, Sistema de Recomendação) deseja visualizar.
- O código usa blocos if/elif para renderizar o conteúdo da aplicação selecionada.

---

5. Seções da Aplicação (Detalhes por Bloco)

5.1. Previsão de Tráfego Urbano (if app_selection == "Previsão de Tráfego Urbano":)

Esta seção exibe o desempenho de um modelo de regressão para prever o tempo de viagem.

- **st.title(" 🚦 Previsão de Tráfego Urbano")**: Título da seção.
- **Carregamento do Modelo**:
    - model = joblib.load("best_traffic_model.joblib"): Carrega o modelo de previsão de tráfego pré-treinado.
    - model_info = joblib.load("best_traffic_model_info.joblib"): Carrega um dicionário com o nome do modelo e suas métricas salvas.
    - Tratamento de erros (try-except) caso o modelo não possa ser carregado.
- **Preparação dos Dados (para inferência e avaliação)**:

- expected_cols: Lista de colunas esperadas pelo modelo treinado. Isso é crucial para garantir que os dados de entrada para o modelo tenham a mesma estrutura (mesmas colunas, mesma ordem) que os dados usados no treinamento.
- X = traffic_data.drop('travel_time', axis=1): Separa as features do dataset de tráfego.
- X = pd.get_dummies(X): Aplica One-Hot Encoding às colunas categóricas restantes em X.
- **Garantia de Colunas Consistentes**: O loop for col in expected_cols: e a linha X = X[expected_cols] garantem que X tenha exatamente as colunas esperadas pelo modelo, adicionando colunas faltantes com valor 0 (o que é comum em One-Hot Encoding quando uma categoria não está presente no conjunto de dados atual, mas esteve no treino) e reordenando-as.
- y = traffic_data['travel_time']: Separa a variável alvo.
- X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42): Divide os dados. **Atenção**: Embora o modelo tenha sido treinado externamente, essa divisão é feita aqui para que as métricas de avaliação (mse, rmse) possam ser calculadas em um conjunto de teste consistente. O X_test e y_test gerados aqui simulam o que o modelo veria em um ambiente de teste.

- **Previsões e Métricas**:
  - y_pred = model.predict(X_test): O modelo carregado faz previsões no conjunto de teste.
  - mse = mean_squared_error(y_test, y_pred): Calcula o Erro Quadrático Médio.
  - rmse = np.sqrt(mse): Calcula a Raiz do Erro Quadrático Médio.
- **Exibição no Streamlit**:
  - Mostra o nome do modelo e as métricas salvas (model_info).
  - Exibe o MSE e RMSE calculados.
  - **Gráfico de Previsão vs. Real**: Usa st.line_chart para mostrar as primeiras 100 previsões comparadas aos valores reais, permitindo uma visualização rápida do desempenho.

- **Gráfico de Distribuição dos Erros**: Usa sns.histplot para plotar um histograma dos erros (y_test - y_pred). Uma distribuição centrada em zero indica que o modelo não tem um viés sistemático.

5.2. Classificação de Saúde Pública (elif app_selection == "Classificação de Saúde Pública":)

Esta seção avalia um modelo de classificação que prevê o nível de risco em saúde pública.

- **st.title("Classificação de Saúde Pública")**: Título da seção.
- **Carregamento de Modelos**:
  - model = joblib.load("best_health_model.joblib"): Carrega o modelo de classificação de saúde.
  - label_encoder = joblib.load("label_encoder_health.joblib"): Carrega o LabelEncoder usado para transformar os rótulos de risco de volta aos seus nomes originais.
  - model_info = joblib.load("best_health_model_info.joblib"): Carrega informações sobre o modelo (nome, hiperparâmetros).
  - Tratamento de erros.
- **Exibição de Informações do Modelo**: Mostra o nome do modelo e os melhores hiperparâmetros.
- **Preparação dos Dados (para avaliação)**:
  - data = pd.read_csv("improved_health_data.csv"): Recarrega os dados de saúde.
  - data['risk_level'] = data['risk_level'].str.lower(): Normaliza os rótulos para minúsculas.
  - data['risk_level'] = label_encoder.transform(data['risk_level']): Codifica os rótulos de risco usando o LabelEncoder carregado.
  - X = data.drop('risk_level', axis=1): Separa as features.
  - y = data['risk_level']: Separa a variável alvo.
  - scaler = StandardScaler(): Inicializa o scaler.
  - X_scaled = scaler.fit_transform(X): Padroniza as features. **Atenção**: O fit_transform no conjunto completo X aqui pode não ser o ideal para uma avaliação *geral* do modelo se ele foi treinado em um subconjunto. Idealmente, o scaler também deveria ser salvo e

carregado, e apenas transform aplicado aos dados. No entanto, para fins de demonstração da performance *geral* do modelo nos dados disponíveis, é aceitável.

- **Previsões e Métricas**:
    - y_pred = model.predict(X_scaled): O modelo faz previsões nos dados padronizados.
    - acc = accuracy_score(y, y_pred): Calcula a acurácia.
    - report = classification_report(y, y_pred, target_names=label_encoder.classes_, output_dict=True): Gera o relatório de classificação como um dicionário.
    - cm = confusion_matrix(y, y_pred): Gera a matriz de confusão.
- **Exibição no Streamlit**:
    - Mostra a acurácia geral.
    - Exibe o relatório de classificação em formato de tabela.
    - **Matriz de Confusão**: Usa sns.heatmap para visualizar a matriz de confusão, com rótulos amigáveis.
    - **Gráfico de Precisão por Classe**: Um gráfico de barras que exibe a precisão para cada classe de risco.
    - **Gráfico de Distribuição Real vs. Previsto**: Compara a distribuição dos rótulos reais e previstos, ajudando a identificar desequilíbrios ou vieses nas previsões do modelo.

5.3. Análise de Sentimentos (elif app_selection == "Análise de Sentimentos":)

Esta seção demonstra um modelo de classificação de sentimento.

- **st.title("🧠 Análise de Sentimentos da População")**: Título da seção.
- **Carregamento de Modelos**:
    - pipeline = joblib.load('sentiment_model.joblib'): Carrega o Pipeline completo (TF-IDF + Classificador) do modelo de sentimento.
    - label_encoder = joblib.load('label_encoder.joblib'): Carrega o LabelEncoder para os sentimentos.
    - Tratamento de erros.
- **Preparação dos Dados (para avaliação)**:

    - sentiment_data['text_clean'] = sentiment_data['text'].apply(preprocess): Aplica a função de pré-processamento de texto (definida no início do script) aos dados de sentimento.
    - sentiment_data = sentiment_data.dropna(subset=['text_clean']): Remove linhas onde o texto limpo pode ter se tornado vazio.
    - X = sentiment_data['text_clean']: Separa o texto limpo como feature.
    - y = label_encoder.transform(sentiment_data['sentiment']): Codifica os rótulos de sentimento.
- **Previsões e Métricas**:
    - y_pred = pipeline.predict(X): O pipeline faz as previsões (ele se encarrega da vetorização e classificação).
    - Calcula e exibe acurácia e relatório de classificação.
- **Exibição no Streamlit**:
    - Mostra a acurácia e o relatório de classificação.
    - **Matriz de Confusão**: Visualiza a matriz de confusão para o sentimento.
    - **Gráfico de Métricas por Classe**: Exibe precisão, revocação (recall) e F1-score por classe.
    - **Palavras Mais Frequentes por Sentimento**: Permite ao usuário selecionar um sentimento e ver as 10 palavras mais comuns associadas a ele, dando insights sobre os dados.
    - **Parâmetros e Configurações do Modelo**: Tenta exibir os parâmetros do TfidfVectorizer e do classificador dentro do pipeline, o que é ótimo para transparência e depuração.

5.4. Sistema de Recomendação (elif app_selection == "Sistema de Recomendação":)

Esta seção implementa e demonstra um sistema de recomendação baseado em filtragem colaborativa User-Based, com a capacidade de adicionar novos usuários.

- **st.title("Sistema de Recomendações de Serviços Públicos")**: Título da seção.
- **Entrada de Novo Usuário**:

    - Permite ao usuário da interface adicionar um novo usuário e registrar suas avaliações para serviços específicos através de st.text_input e st.multiselect/st.slider.
    - Quando o botão "Adicionar usuário" é clicado, os novos dados são concatenados ao recommendation_data DataFrame.
- **Reconstrução da Matriz Usuário-Serviço e Similaridade**:
    - Após a possível adição de um novo usuário, o código **recalcula** a matriz de interesse ponderado (weighted_interest = rating * frequency).
    - Recalcula a user_service_matrix usando pivot_table.
    - Recalcula as matrizes de similaridade (user_similarity_df, user_similarity_normalized_df) usando cosine_similarity. **Importante**: Isso significa que o "modelo" de recomendação não é salvo e carregado, mas sim recalculado dinamicamente a cada interação/adição de usuário, o que é viável para datasets pequenos, mas não escalável para grandes volumes de dados.
- **Função recommend_services (Atualizada para o Streamlit)**:
    - É a mesma lógica da função original de recomendação.
    - Inclui mensagens st.warning e st.info para feedback ao usuário sobre o status da recomendação (usuário não encontrado, já utilizou todos os serviços, fallback para populares).
- **Interface de Recomendação**:
    - selected_user = st.selectbox(...): Permite ao usuário selecionar um usuário existente (ou o recém-adicionado) para obter recomendações.
    - Ao clicar em "Recomendar serviços", exibe as recomendações geradas.
- **Visualizações**:
    - **Mapa de Calor da Similaridade**: Mostra a similaridade entre usuários, visualizando os "vizinhos" de gosto.
    - **Popularidade dos Serviços**: Exibe um gráfico de barras dos serviços mais populares, que é usado como fallback nas recomendações.

---

6. Observações Gerais sobre a Implementação

- **Modularidade:** A aplicação é bem estruturada com módulos para cada tipo de problema de ML, tornando-a fácil de navegar e entender.
- **Uso de Modelos Persistidos (joblib):** Para as seções de Tráfego e Saúde/Sentimento, os modelos são carregados de arquivos joblib. Isso significa que os modelos foram treinados e salvos *separadamente* e a aplicação Streamlit serve apenas para demonstração/inferência, o que é uma boa prática.
- **Recálculo Dinâmico (Recomendação):** A seção de recomendação recalculou a matriz de similaridade em tempo real. Embora funcional para pequenas demonstrações, isso seria um gargalo de performance para um sistema de recomendação real em larga escala.
- **Apresentação de Métricas e Visualizações:** Para cada módulo, o código não apenas faz previsões, mas também apresenta métricas de avaliação e visualizações relevantes (gráficos de dispersão, matrizes de confusão, importâncias de features, etc.), o que é excelente para demonstrar a performance do modelo.
- **Feedback ao Usuário:** O uso de st.success, st.error, st.warning, st.info e outros elementos de Streamlit oferece feedback claro ao usuário sobre o status da aplicação e dos processos.