ChatGPT

Data Science Project Guides: SmartSave and Loan Default Prediction

Overview: These two project guides break down each project into phases with actionable steps. Each phase includes a checklist of tasks, focusing on Python-based tools (with SQL, visualization, and simple web frameworks where appropriate). Both projects are designed as portfolio-ready pieces to showcase end-to-end data science skills, with an emphasis on clean code structure and GitHub best practices for internship readiness.

SmartSave: Grocery Price Comparison Tool

SmartSave is a data engineering and analytics project that **scrapes grocery store prices, cleans the data, stores it in a database, and provides a simple frontend** for comparing item prices across multiple stores. It demonstrates skills in web scraping, data cleaning, database use, and building a user-facing dashboard for insights. The workflow is broken into five phases (Planning, Data Collection, Processing, Reporting/Dashboard, and GitHub Delivery) to mirror a typical data pipeline [1].

Phase 1: Planning

- **Define Project Scope & Goals:** Clearly outline what SmartSave will do – e.g. *"scrape product prices from X grocery store websites and enable price comparison by item across stores."* Identify which stores or supermarkets to include and what data points (product name, unit price, store name, date, etc.) you need [2].
- **Data Source Analysis:** For each target store, investigate how product information is presented (HTML structure or API). Check if the site has public APIs or if web scraping is needed. Ensure scraping these sites is allowed (review the terms of service or `robots.txt`).
- **Tool Selection:** Decide on tools and libraries. Plan to use Python for scraping (e.g. `requests` and `BeautifulSoup` for static pages, or Selenium for dynamic content), and a database for storage (SQLite for simplicity or PostgreSQL for a more robust solution). Using a lightweight database will make the data easy to query for the dashboard [3].
- **Data Model Design:** Sketch out a schema for storing the data. For example, a single table with fields: `item_id` (or name), `store`, `price`, `unit` (if applicable), and `date_collected`. This will help maintain consistency when aggregating data from multiple stores.
- **Project Timeline & Milestones:** Break the project into phases with rough timelines. Plan the order of work: e.g. complete one store's scraper and database load first as a prototype, then extend to others; only then build the frontend.
- **Success Criteria:** Define what a successful project looks like – e.g., *"The app can show a user the current prices of a given product at all scraped stores."* Also identify stretch goals (like scheduling regular price updates or adding user features) but focus on core functionality first.

Phase 2: Data Collection (Web Scraping)

- **Set Up Scraping Environment:** Create a Python script or notebook for web scraping. Install required libraries (`requests`, `beautifulsoup4`, etc., and possibly `selenium` with a driver if some sites heavily use JavaScript). Test that you can fetch a page from each target site.
- **Scrape Each Site's Data:** Write scraping routines for each grocery store website. Parse the HTML to extract product name and price (and other info like product size/quantity if needed). Each website will have a unique structure – you may need to find specific HTML tags or classes that contain the product listings [4] [5]. For example, find the container that lists products and loop through its children to get name and price for each item.
- **Politeness & Error Handling:** Implement delays (`time.sleep()`) between requests or use an exponential backoff to avoid overwhelming the site. Set a custom User-Agent header so your scraper is less likely to be blocked. Include try/except blocks to handle missing elements or connection issues gracefully (e.g., skip a product if a field is missing, and log the error).
- **Iterate and Verify:** Run the scraper on a small sample (e.g., one category or one page) and print some results to verify correctness. Ensure the data makes sense (e.g., prices are captured as numbers, product names are complete). Manually spot-check a few items against the website to confirm accuracy.
- **Collect Data from Multiple Pages:** If product listings span multiple pages, make sure your scraper navigates through all pages (e.g., by altering URL page parameters or clicking "next" links if using Selenium). Append results for all pages.
- **Store Raw Data Temporarily:** As you scrape, accumulate results in a list of dictionaries or a pandas DataFrame. Save the raw scraped data to a CSV file as a backup and for transparency. This raw data snapshot can be useful for cleaning or if the scraping needs to be repeated.
- **Scrape Multiple Stores:** Repeat the process for each target store. It might be useful to write each store's scraper as a separate function or script (for modularity). Aim to gather a comparable set of fields from all stores (name, price, etc.), so the data can eventually be combined or compared.
- **Initial Data Volume Check:** Note how many products you've scraped from each store and roughly how many unique items (by name) that represents. This gives an idea of the dataset size and helps plan database storage.

Phase 3: Data Processing (Cleaning and Database)

- **Data Cleaning:** Load the scraped data (from CSV or DataFrame) and clean it with pandas. Normalize the pricing data – e.g., remove currency symbols or units and convert price strings to numeric values. Strip whitespace from product names, and standardize the casing or naming convention if needed (e.g., you might want all names in lowercase for easier matching). Handle any missing or extreme values (if some prices failed to scrape or are clearly outliers).
- **Identify Common Products:** Since the goal is to compare prices of the *same* item across stores, consider how to match products from different retailers. Exact name matches might be tricky (stores may use slightly different naming). As a starting point, you could compare by keywords (e.g., by item name without brand or size) or focus on a subset of well-known products. Document this assumption or method. For now, the frontend can use a keyword search to find similar items across stores [6], so perfect matching isn't mandatory in the data cleaning stage.
- **Set Up Database:** Initialize a database to store the cleaned data. For a beginner-friendly route, use **SQLite**, which doesn't require a server and stores data in a file. (This can later be swapped to PostgreSQL if desired without changing much code, thanks to SQL syntax similarity.) Design a table (e.g., `prices`) with columns for product name, store, price, and date.

- **Load Data into DB:** Use Python to insert the cleaned data into the database. You can use pandas `DataFrame.to_sql()` for SQLite or use SQLAlchemy for more control. Ensure primary keys or indexes make sense (for example, an auto-increment ID or a composite key of (product, store, date)). After insertion, run a few test queries (in Python or using a DB browser) to ensure data integrity – e.g., count records per store, or find a sample product across stores to see if data was loaded correctly.
- **Data Querying Test:** Practice a couple of SQL queries that you'll need for the app. For instance, *"find all prices for 'milk' across stores ordered by price"* or *"get the latest price of each product per store"* if you plan to show only recent data. This will confirm that your data model supports the queries needed for comparison.
- **Maintain Data Hygiene:** If any duplicates or anomalies were found during cleaning (e.g., identical product entries), decide how to handle them (maybe keep only the latest price per item per store). If scraping was done over multiple days, you might have historical prices – that's useful for analysis, but you might choose to filter to the most recent date for the comparison dashboard, or even build a simple time-trend view.
- **Efficiency Consideration:** Ensure the data volume is manageable. If you scraped thousands of items, the SQLite database file might be a few MBs, which is fine. If it's much larger, consider whether you need all data or if you should limit categories. Keep the dataset small enough to easily work with for a demo.
- **Backup Clean Data:** Export the cleaned combined dataset to a CSV as well. This can serve as a static data source for the dashboard in case the database access is an issue, and it's a good artifact to include in the project (if file size is reasonable).

Phase 4: Reporting/Dashboard (Frontend)

- **Choose a Presentation Method:** Decide how you will enable users (or recruiters) to interact with the price comparison. A simple approach is to create an interactive **Streamlit** app, which can be done entirely in Python with minimal web development [7]. Alternatively, you could build a small **Flask** web application with an HTML template for results, or even just produce a Jupyter Notebook with analysis and visualizations. Given your skillset, **Streamlit is recommended** for a quick, polished result (it allows dropdowns, search boxes, and displays data frames/charts easily).
- **Implement Item Search:** Design the front end around a keyword search or item selector. For example, provide a text input for the user to type an item name (like "milk"). When the user submits, query the database for any products matching that keyword (across all stores), and display the list of stores and prices for those items [8]. This achieves the core goal of comparing prices of the same or similar product across different retailers.
- **Display Results Clearly:** Show the comparison in a user-friendly format. This could be a table (with columns for store, product name, price, date) or a bar chart where each store's price is a bar for easy comparison. Highlight the lowest price, for example, by sorting or using a different color in a chart. If using Streamlit or Flask, ensure the output is nicely formatted (Streamlit can display a pandas DataFrame directly or use `st.bar_chart` for quick visuals).
- **Additional Insights (Optional):** Enhance the dashboard with a few extra features if time permits. For instance, you could add a filter for date (to compare current prices only), or a dropdown to pick a specific product from a list instead of free-text search (you could pre-populate it with popular items found in the data). Another idea is a summary view – e.g., showing the number of items where each store is cheapest, or a visualization of price distribution per store. Keep these as stretch goals after the basic functionality works.

- **Testing the Dashboard:** Run your app locally and test a variety of item searches. Verify that it returns results from all stores and that the prices make sense. This testing might reveal data issues (e.g., one store's price scraped as "$5 per 100g" vs another as "$10 per pack" – which aren't directly comparable). If such cases arise, document them or refine the data processing to handle units better. For the demo, you might restrict to items where comparison is apples-to-apples.
- **Deployment (Optional):** If using Streamlit, consider deploying the app to Streamlit Cloud via your GitHub repo, so that recruiters can interact with it live. This involves pushing your code to GitHub and then sharing the app link. If that's not feasible, you can record a short video/gif of the app in action or include screenshots in your README to demonstrate the working dashboard.
- **User Guidance:** Ensure the UI or output has some quick instructions or labels. For example, if it's a text input, prompt with "Enter a product name to compare prices." Small touches like this make your project more polished and accessible.

Phase 5: GitHub Delivery

- **Organize the Repository:** Structure your project folder logically. Separate your code and data. For example, have a `scraping/` directory for the web scraping scripts, a `data/` folder for raw and cleaned datasets (or better, exclude raw large data from Git and provide download instructions), and an `app/` folder for the Streamlit or Flask app code. A structured layout helps others navigate your project [9] [10].
- **Clean Up and Comment Code:** Before publishing, review your code for clarity. Remove any leftover debug print statements or unused functions. Add comments and docstrings to explain non-obvious parts. This makes it easier for someone reading your code (like an interviewer) to follow your logic. Ensure your functions and variables have meaningful names.
- **README – Project Overview:** Create a **detailed README.md** at the root of the repo. This should serve as a standalone description of the project. Include a brief introduction to SmartSave's purpose (e.g., *"A Python project that scrapes grocery prices and lets users compare them across stores in an interactive dashboard."*). List the tools/libraries used (Python, BeautifulSoup, pandas, SQLite, Streamlit, etc.) [11]. Provide instructions on how to install any requirements (perhaps via a `requirements.txt`) and how to run the application or notebooks [12]. The README should also mention where the data comes from (the names of the grocery websites, without linking to any proprietary content) and any considerations (like "*data is scraped on XYZ date and may not be updated*").
- **README – Results and Demo:** Include screenshots or even a GIF of your running dashboard in the README to immediately show the outcome. For example, display an image of a price comparison result for a sample item. This makes the project *visual* and enticing. You can also summarize interesting findings, e.g., "*We found Store A had the cheapest average prices on produce, while Store B often had the lowest prices on dairy.*" (if such patterns exist). Describing these insights helps "sell" the project by showing it's more than just coding – it produced useful information [13].
- **Ensure Reproducibility:** Provide a `requirements.txt` file listing all Python packages (with versions) needed to run the code. This allows others to install dependencies easily. If using a database like SQLite, you might include the pre-populated `.db` file (if it's not too large) in the repo for convenience. Otherwise, include instructions or a script to create and populate the database from the CSV (so the user or recruiter can reproduce the database on their machine).
- **Version Control Hygiene:** Double-check that you are not committing any sensitive information (like personal credentials or API keys – probably not applicable here unless you used some secret API). Use a `.gitignore` to exclude unnecessary or large files – for instance, caching directories, the virtual environment, or large raw data dumps. Only track the essential source code, small sample

data, and documentation [14]. If the grocery data is large or scraped under terms that disallow redistribution, do not include it directly; instead, note in the README how to run the scraper to get the data or provide a limited sample [14].

- **Licensing and Attribution:** Consider adding an open-source license if you want (MIT is a simple default) so others know they can use your code. Also, credit any sources – for example, "*Prices data was collected from public web pages of Store X, Y, Z*". This shows professionalism and respect for data sources.
- **Final Verification:** Pretend you are a newcomer to the project – follow your own README steps on a fresh machine (or environment) to ensure everything works as described. This QA step helps catch missing instructions or setup steps. Once verified, commit and push all final changes with clear commit messages (e.g., "Add data cleaning script and update README with usage instructions"). Now your SmartSave project is ready to share on your resume and with recruiters!

Loan Default Prediction Project Guide

The **Loan Default Prediction** project is a machine learning portfolio piece using a public dataset (e.g., from Kaggle) to predict whether a borrower will default on a loan. This showcases skills in data cleaning, statistical analysis, and predictive modeling using Python's data science stack. You will walk through defining the problem, exploring and processing the data, building a classification model (likely using scikit-learn), and presenting the results. Emphasis is on proper modeling techniques and insightful communication of findings. We break this project into six phases: Planning, Data Collection, Processing, Modeling, Reporting, and GitHub Delivery.

Phase 1: Planning

- **Project Understanding & Goal:** Start by clearly articulating the purpose: e.g., *"Predict whether a loan will default (not be repaid) based on borrower information."* In real terms, this means identifying high-risk loans before they are issued. Define the output (probably a binary classification: default or not) and how success will be evaluated (accuracy, F1-score, AUC, etc., with attention to handling class imbalance if defaults are rare).
- **Choose a Dataset:** Locate a suitable public dataset. A popular choice is a **Kaggle loan default dataset** (for example, from Lending Club or a similar financial dataset). Ensure the dataset has the features you need – common ones include borrower income, loan amount, credit history, etc. – and a target column indicating default or not [15]. If using Kaggle, you may need to sign up and download the data manually or use the Kaggle API.
- **Assess Domain & Features:** Read any documentation about the dataset to understand feature definitions (e.g., what does each column mean? which are categorical vs numeric?). Consider how each might relate to default risk (this will guide feature engineering later). For instance, *credit score, debt-to-income ratio, past default flag* are likely important predictors. Write down assumptions or expected relationships (e.g., "I expect higher income to correlate with lower default probability").
- **Technical Stack for Modeling:** Plan the tools for each step. You'll likely use **pandas** for data manipulation and **Matplotlib/Seaborn** for exploratory visualizations. For modeling, choose scikit-learn algorithms (e.g., logistic regression for a baseline, and perhaps a decision tree or random forest for a more complex model). Ensure you have or can install any needed libraries like scikit-learn, and maybe **imbalanced-learn** if you plan to address class imbalance with SMOTE or similar.