Data description

The dataset “Risk Factor Prevalence Study, 1980” was obtained from the Australian Data Archive (ADA). The survey documentation is public, available for anyone to read, but to download the data, one needs permission from the ADA. Citing requirements, as well as a copyright and disclaimer can be found below.

After requesting and gaining access to the dataset, I could download 7 files: the documentation and data of the survey from the website:
https://dataverse.ada.edu.au/dataset.xhtml?persistentId=doi:10.26193/BYE1RE

The data type is a survey; clinical data which was collected from Australian residents in capital cities between the ages 25–64 in 1980.
The primary investigator was the above-mentioned Australia, N. H. F. O., and the survey was partially funded by the Commonwealth Department of Health.

Citation Requirement: “Australia, N. H. F. O. Risk Factor Prevalence Study, 1980 [computer file]. Canberra: Australian Data Archive, The Australian National University”.
Rights & Disclaimer:
“Use of the material is solely at the user's risk. The depositor, The Australian National University and the Australian Data Archive shall not be held responsible for the accuracy and completeness of the material supplied.”
Copyright:

The dataset has 169 variables with 5,603 cases, the sas7bdat data file is 7.7 MB. The variables of the dataset follow the questions of the survey. From the 169 variables, I used 34 from the original dataset to predict a heart attack. I added a calculated variable: BMI, since it is a better indicator of obesity than weight and height separately. I used the formula:

$$\text{BMI} = \text{weight (kg)} / (\text{height (m)})^2$$

taken from the CDC website:
https://www.cdc.gov/nccdphp/dnpao/growthcharts/training/bmiage/page5_1.html

I renamed the variables from the question numbers to more specified names. Variable 8, “heart_attack”, from the list below is the target data, and all remaining variables are the feature data.

Variables description:

	Variable name	Description	Values	Data type
0	sex	sex of the patient	1: male 2: female	discrete
1	age	age of the patient	1: 24 – 29 2: 30 – 34 3: 35 – 39 4: 40 – 44 5: 45 – 49 6: 50 – 54 7: 55 – 59 8: 60 – 64	discrete
2	BMI	calculated from weight and height	numeric	continuous
3	chestpain	responder experienced chest pain	1: no 2: yes	discrete
4	chestpressure	esponder experienced chest pressure	1: no 2: yes	discrete
5	diabetes	responder has diabetes	1: no 2: yes	discrete
6	HBP	responder has HBP	1: no 2: yes	discrete
7	angina	responder has angina pectoris	1: no 2: yes	discrete
8	heart_attack	responder had heart_attack	1: no 2: yes	discrete
9	stroke	responder had stroke	1: no 2: yes	discrete

10	highcholesterol	responder has high cholesterol	1: no 2: yes	discrete
11	hightriglyc	responder has high triglyceride levels	1: no 2: yes	discrete
12	sleep_aid	frequency in which the responder takes sleeping aids	1: every day 2: a few days a week 3: once a week 4: occasionally 5: rarely 6: never	discrete
13	sedatives	frequency in which the responder takes sedatives or tranquilizers	1: every day 2: a few days a week 3: once a week 4: occasionally 5: rarely 6: never	discrete
14	vitamins	how often the responder takes vitamins or mineral supplements	1: every day 2: a few days a week 3: once a week 4: occasionally 5: rarely 6: never	discrete
15	sleep_hours	the number of hours the responder usually sleeps in a night	1: 5 hours or less 2: 6 hours 3: 7 hours 4: 8 hours 5: 9 hours or more	discrete
16	sleep_quality	how the responder describes their normal sleep	1: poor 2: fair 3: good	discrete
17	add_salt	how often the responder adds salt to food	1: not at all 2: sometimes 3: only after fasting 4: always	discrete
18	meat	how often the responder eats meat	1: every day 2: most days 3: at least once a week 4: infrequently 5: never	discrete

19	fat	how often the responder eats the fat on meat	1: every day 2: most days 3: at least once a week 4: infrequently 5: never	discrete
20	eggs	how many eggs the responder eats per week	numeric	numeric
21	fat_type	which type of fat the responder eats most often	1: butter 2: polyunsaturated margarine 3: other table margarines 4: I rarely eat any of these 5: I don't eat any of these	discrete
22	alcohol	how often the responder drinks alcohol	1: don't drink alcohol 2: less than once a week 3: on 1 or 2 days a week 4: on 3 or 4 days a week 5: on 5 or 6 days a week 6: every day	discrete
23	num_drinks	Number of drinks the responder consumes per occasion	1: don't drink alcohol 2: 1 or 2 drinks 3: 3 or 4 drinks 4: 5 to 8 drinks 5: 9 to 12 drinks 6: 13 to 20 drinks 7: more than 20 drinks	discrete

24	walking	how often the responder walks for exercise	1: three times or more a week 2: once or twice a week 3: once a month 4: rarely 5: never	discrete
25	cardio	how often the responder exercises vigorously	1: three times or more a week 2: once or twice a week 3: once a month 4: rarely 5: never	discrete
26	sport	how often the responder engages in other sports	1: three times or more a week 2: once or twice a week 3: once a month 4: rarely 5: never	discrete
27	hobby	other physical activities, like hobbies	1: three times or more a week 2: once or twice a week 3: once a month 4: rarely 5: never	discrete
28	work	how much time the responder spends walking while working	1: practically all 2: more than half 3: about half 4: less than half 5: almost none	discrete
29	SYS1	systolic blood pressure first measure	numeric	continuous
30	SYS2	systolic blood pressure second measure	numeric	continuous

31	DIA1	diastolic blood pressure first measure	numeric	continuous
32	DIA2	diastolic blood pressure second measure	numeric	continuous
33	SERCHOL	serum cholesterol level	numeric	continuous
34	HDLCHOL	HDL cholesterol level	numeric	continuous
35	TRIGLYC	triglyceride levels	numeric	continuous

Software

I used the Jupyter Notebook with Python 3.7 for the model building part, and FastAPI and Streamlit for the web application. Docker was used to build images for the web server which is an EC2 instance on AWS. Zoom will be used for the video presentation. Python libraries: numpy, pandas, math, matplotlib, pyplot, seaborn, pyreadstat, sklearn, scipy, tensorflow, pickle, uvicorn, fastapi, streamlit.

Analyses

First, I got familiar with the dataset. I thoroughly read the survey documentation and explored the variables of the dataset. After choosing the variables I planned to work with, I started the data exploration with descriptive statistics and data visualization. Graphs helped to find and understand relationships between the variables. We also checked the distribution and skewness of the data. Boxplots are useful for detecting outliers, histograms present distribution, and scatter plots show relationships between two continuous variables.

Next I cleaned and prepared the data for the ML algorithms. I addressed the missing values with imputation. Because the dataset is relatively small, I didn’t drop any missing

values, but for both categorical and continuous features, I used KNN Imputer. For the categorical features, missing values were “nan”, while for the continuous data it was zero. Therefore, I replaced zeros with “nan” for easy imputation. The categorical features first were ordinal encoded then imputed and finally I got the labels back by reversing ordinal encoding. After the continuous features were imputed too, I joined all features back together into one dataframe. After I had no missing values, I continued data exploration with the info(), describe(), and corr() methods. Since I wanted to predict heart disease, I compared patients' data with and without heart attack using boxplots, barplots, scatterplots and cat plots with the seaborn library. Finally, I scaled the continuous data using the Standard Scaler from scikit-learn preprocessing library.

After I had a clean and preprocessed dataset, I was ready to move on to model building. I created two dataframes: features and target. The target data frame contains only one column, the feature I want to predict: “heart_attack”. The features dataframe has the remaining 33 features as described above in the Data Description part.
Then, I split the data into training data and test data using train_test_split and 20% test set size. I used the test set after the hyperparameters were fine-tuned.

ML Model Training:

Logistic Regression: with penalty, C, solver
Support Vector Classification: C, kernel, degree, gamma, coef0
Decision Tree Classifier: criterion(default), splitter(default), max_depth, min_samples_split, min_samples_leaf (influences only the feature importance list), max_features
Random Forest Classifier: criterion, n_estimators, max_depth, min_samples_split, min_samples_leaf, max_features, max_leaf_nodes
KNN: n_neighbors, weights, algorithm, leaf_size, metric - default worked best
ANN with keras and tensorflow.

Logistic Regression: penalty, C, solver parameter were used.
The logistic regression model is a binary classifier, uses the S-shaped logistic function: estimates the probability of the positive class. If the probability is greater than 0.5, the instance belongs to the positive class, if it is less, then the model predicts that it belongs to the negative class. To calculate the probability, the model computes the weighted sum of the input features (plus a bias term), and it outputs the logistic of this result. Since this value is always between 0 and 1, the model then easily computes the probability.

First, I trained a Logistic Regression model with default values and checked the accuracy, precision and recall scores. These scores were the same, and relatively high. We also used the trained model on the training set to check if the model is overfitted. Since the metrics came back very close to the test set scores, I can conclude that the model was not overfitted. I plotted a confusion matrix to check false positives and false negatives in the prediction. The model predicted 11 positive cases for heart attack, but only 4 were true positives. 7 were false positives, and 13 were false negatives.
After that I performed a grid search to find the best parameters for penalty, C, solver, starting with broad intervals then narrowing them in the following grid searches. After 2 grid searches I found the best parameters: 'C=206.913808111479, penalty= l2, solver= liblinear. We trained a new model with the best parameters and checked the generalization error with the same metrics and MSE. After grid search, the confusion matrix was the same.
I also checked the feature importances to understand the model better. The top six features in predicting a heart attack are: angina, chest pain, stroke, chest pressure, high triglyceride levels and high cholesterol.
Best parameters:
C = 207, penalty = 'l2', solver = 'liblinear'

Support Vector Classification: with C, kernel, degree, gamma, coef0
The second model was a vector classification model. The Support Vector Classifier is a linear model, with an algorithm that creates a line or a hyperplane to separate the data into classes. Since the gamma value is low in the best model, that means that even the data points far away from the boundary get considerable weight and the curve is more linear.
After the grid search, I trained a model with the best parameters, and checked the performance metrics. Accuracy, precision, and recall scores came back the same, 98.48 and MSE=0.015. The model predicted 8 positive cases for heart attack, 4 true positives and 4 false positives. There were also 13 false negative cases.
Then I performed dimensionality reduction with PCA to see how the results change while preserving 95% variance. The performance metrics were the same after the PCA, but the confusion matrix was slightly different. Now the model predicted zero positive cases, and 17 false negatives.
Best parameters:
C=5
coef0=1
gamma=0.005
kernel='poly'

Decision Tree Classifier:
The Decision Tree algorithm splits the nodes to have the lowest gini scores. If the model is not constrained, it splits nodes until it reaches the lowest gini scores. Then it stops splitting further and the node becomes a leaf. In practice, the gini score is calculated for every node, then the weighted average of these scores becomes the gini of the given feature/question. Then after every features' gini score is calculated this way, the feature with the lowest gini score will be the next feature that splits the dataset.
The decision tree classifier model was trained with criterion(default), splitter(default), max_depth, min_samples_split, min_samples_leaf (influences only the feature importance list), and max_features. However, I first checked how a default model performs: the metrics were slightly lower than with the previous models: accuracy, precision, and recall scores = 0.97056. The model predicted 22 positive cases, from which 19 were false positives. There were also 14 false negative cases. The feature importance list was a little different from the logistic regression model: the top six features were angina, BMI, systolic blood pressure, walking time during work, number of alcoholic drinks per day, and sleep hours per night.
After performing grid search, I trained a new model with the best parameters and checked the performance metrics: they slightly improved: accuracy score, precision, and recall scores = 0.984 and MSE=0.016. From the confusion matrix I learned that the model predicted 3 positive cases from which 2 were false positives. And there were also 16 false negatives. After further tuning the model by hand, I found another model which performed slightly better: accuracy, precision, and recall scores were 0.9866. The model predicted 8 positive cases, but 3 were false positives. There were also 12 false negative cases. The feature importances also changed.
The best parameters were:
max_depth=8
max_leaf_nodes=30
min_samples_split=20
max_features=12
min_samples_leaf=12
criterion(default)=gini
splitter(default)=best

Random Forest Classifier: with criterion, n_estimators, max_depth, min_samples_split, min_samples_leaf, max_features, max_leaf_nodes
The random forest model is a tree based classifier that splits nodes to minimize the gini score or entropy. While one single decision tree trains fast, a random forest takes much longer as it grows many random trees.
The first model with default parameters predicted 2 true positive cases and 15 false negatives. The metrics were (accuracy, precision, recall scores): 0.9866.

After grid search, the scores slightly decreased to 0.9848 and the MSE=0.151.
The feature importances was similar to the best decision tree model.
The best parameters are:
max_depth=32
max_leaf_nodes=18
min_samples_leaf=2
n_estimators=100
criterion='gini'

KNN Classifier: For this algorithm, n_neighbors, weights, algorithm, leaf_size, metric parameters were used, and the default parameters gave the same result as the best parameters. The KNN algorithm classifies a new datapoint by comparing it to a given number of its nearest neighbors. The models predicted 2 positive cases from which 1 was false positive. There were 16 false negatives as well. The accuracy, precision, recall scores were 0.9848. The best parameters were:
n_neighbors=5
algorithm= 'auto'
leaf_size=10
metric='minkowski'
weights= 'uniform'

ANN with keras and tensorflow:
For the ANN model, the training data was further divided into training and validation sets. 450 data points were used for cross validation. After fine tuning the parameters, the following model performed the best: I trained a model with four dense layers, 1 input layer, 2 hidden layers and 1 output layer, with ‘relu’ and ‘softmax’ activations. The layers have 5, 3, 3, and 1 neurons. We used 400 epochs with early stopping and with patience=10. The MSE is computed on the validation samples. I also used learning curve graphs to check if the validation loss got close to the training loss. The accuracy score was 0.9848 and MSE=0.0152.

ML Model Results and Discussion:

The hyperparameters were fine-tuned using grid search to find the best parameters. Finding the best parameters to constrain the models helps prevent overfitting. Our goal when training ML models is not to get the best performance metric scores, but to have a model that generalizes well on new, unseen data. I also added some visuals to help understand the results: tree plots and confusion matrices. I compared the performance metrics (accuracy, precision, recall, MSE) which is summarized in the table below:

Models	Accuracy	Precision	Recall	MSE
Logistic Regression	0.9822	0.9822	0.9822	0.0152
SVM Classifier	0.9848	0.9848	0.9848	0.0152
Decision Tree Classifier	0.9839	0.9839	0.9839	0.0161
Random Forest Classifier	0.9848	0.9848	0.9848	0.0152
KNN Classifier	0.9848	0.9848	0.9848	0.0152
ANN	0.9848			0.0152

As we can see, the performance metric scores are identical or very similar in every model. However, the confusion matrices are different and offer additional insight: I chose the SVM Classifier model for the web application since it predicted the least numbers of false negatives and false positives. I think when we try to predict a deadly disease, it is important to keep the false negative numbers to a minimum. In this way, we won't miss those patients who might be in danger of having a heart attack.
While the scores were very similar, some of the models did not predict any positive cases. I think this is due to the fact that the dataset contained only 108 positive cases in the 5603 data points. Considering this relatively low number, I think most models had difficulty learning to predict positive cases. We could probably improve this result by combining several years of data into one dataset, and dropping some of the negative cases so the ratio of positive and negative cases could improve.

Models	True Positive	True Negative	False Positive	False Negative
Logistic Regression	4	1097	7	13
SVM Classifier	4	1100	4	13
Decision Tree Classifier	1	1102	2	16
Random Forest Classifier	0	1104	0	17
KNN Classifier	1	1103	1	16
ANN	1	1103	1	16

After I picked the best model for the web application, I used the pickle library to save the model for FastAPI.

ML Model User-Interface:

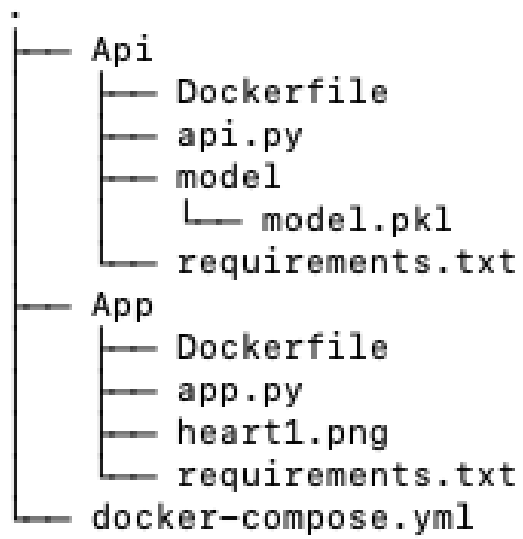
The final step was to build a web application using FastAPI and Streamlit, and to launch it on AWS.
I planned to deploy the machine learning model in Docker containers and serve them as microservices. The advantage of microservices is that the project can be divided into smaller components and they can be deployed independently. This gives us more flexibility to maintain the software: every microservice can be updated and optimized separately which in the end saves resources. In order to use the microservice architecture, we need containerization. I used Docker containers to package the frontend and the backend separately with all their dependencies and isolated environments. Dockerfiles are used to define the containers and build Docker images. Then the images are used on a server as blueprints for the containers. The containers communicate via an API: the frontend container sends the requests - inputted by the user - and the backend container which contains the trained model sends predictions back to the frontend. We also need Docker-compose, which is a tool that makes it possible for the two containers to communicate with each other.

First, I built the backend of the application with FastAPI. We need to import the needed libraries unicorn, pickle, sys, FastAPI, and pydantic. Then we use the BaseModel from pydantic to declare the variables and their data type. Next, I created an object from the FastAPI class. Then I to built two endpoints: one for the input data and one for the prediction. Then input data from the frontend was processed and forwarded to the model for prediction. The code for the backend is found in the “api.py” Python file. We can use uvicorn to serve the backend by the command: “uvicorn api:api” in the terminal. FastAPI has an automatically generated documentation which is also useful for initial testing of the model.

For building the frontend, I used Streamlit which provides a simple but user friendly interface. First, we import the necessary libraries: streamlit, pandas, requests, json, and pydantic. We use the BaseModel from pydantic to declare the variables and their data type. Next, we get input data from the streamlit frontend. Then the input data is combined into a JSON object and a post request is made to the “prediction” endpoint. There is also a “Show Details" checkbox that provides some explanations on how the app works. The code for the frontend is in the app.py. To run the frontend, we use the command: “streamlit run app.py.

Now that both the frontend and backend have been built, we can test our application locally by running them from the terminal: http://localhost:8501.

The next step is containerizing the backend and frontend separately with Docker. We need a different Dockerfile for each container. The Dockerfile is a text file with the commands to build a Docker image. We also need to include the dependencies for each container. The dependencies are in the “requirements.txt” file which was automatically generated by the “pipreqs” package. In the Dockerfile we can define the work directory for the application and copy all the files to the container. I built the images using the “docker build -t <frontend_or_backend_name> .”. I downloaded the Docker desktop app which makes it easy to manage images and containers. I also set up an account in Docker Hub to create a repository and then pushed the images and docker-compose.yml there.
Next step was to create a docker-compose.yml file that coordinates the communication between the frontend and backend containers. In this file we define the services, the working directory, the container and image names and the ports that will be used.
I set up the following file structure:



We run the docker-compose.yml file from the directory which is located in with the command: “docker-compose up”.

To be able to deploy the application on AWS, first I needed to create an account and set up an EC2 instance which is a virtual machine provided by Amazon. The instance is based on the Linux 2 AMI which is a t2.micro instance type. The instance is in the US-East 1 region and I chose the us-east 1a availability zone. After creating the instance, I could choose to get a new private key which Amazon generated.

Then I created a new VPC (virtual private cloud) and chose a private IP block: 10.0.0.0 and then Amazon picked the private IP address 10.0.0.245 for the instance. Amazon calls the public IPs elastic IP, which means it is static. Next I allocated an elastic IP address and associated it with the instance. This IP address is region specific. Amazon

also generated a default DNS name that resolves to this public IP. In the security section, I added two new security rules to open ports 8000 and 8501 for the backend and frontend.
After I set up the instance, I could connect to it with “ssh - <URL with path>”. Next I installed Docker and docker-compose from GitHub. After that, I pulled the images from the Docker Hub repository using the commands:
"docker pull hpiringer/capstone:fastapibackend" and
"docker pull hpiringer/capstone:streamlitfrontend". Then with the “docker-compose up” command I started the application which is now available on the following URL:

http://ec2-52-54-129-72.compute-1.amazonaws.com:8501/