Predicting Strokes

A study of ML and NN in predicting strokes accuracy and readability for clinicians

Conceit of exploration

Taking stroke data and processing it and putting it through traditional ML and a Neural Network to help a Neurology Clinic better screen patients for risk of stroke. We are looking to find an ideal balance between accuracy and readability for our clinic customer.

Pre processing issues

First we started the project on our local host bringing the data into SQL then using SQLAlchemy to query it and clean it with pandas. The goal was to do all of this in Jupyter Notebook then in the same file perform machine learning. But then we realized our systems might not be able to handle the data and moved everything to Databricks. After having difficulty in Databricks we went to Collab.

Data and Cleaning process

Sourced csv from McKinsey Analytics Online Hackathon

Initially cleaned in pandas:

dropped rows with missing values with df.dropna()

Then we used scaler.fit to fit the scaler to the selected numeric columns from the DataFrame.

we transformed the data using scaler.transform()

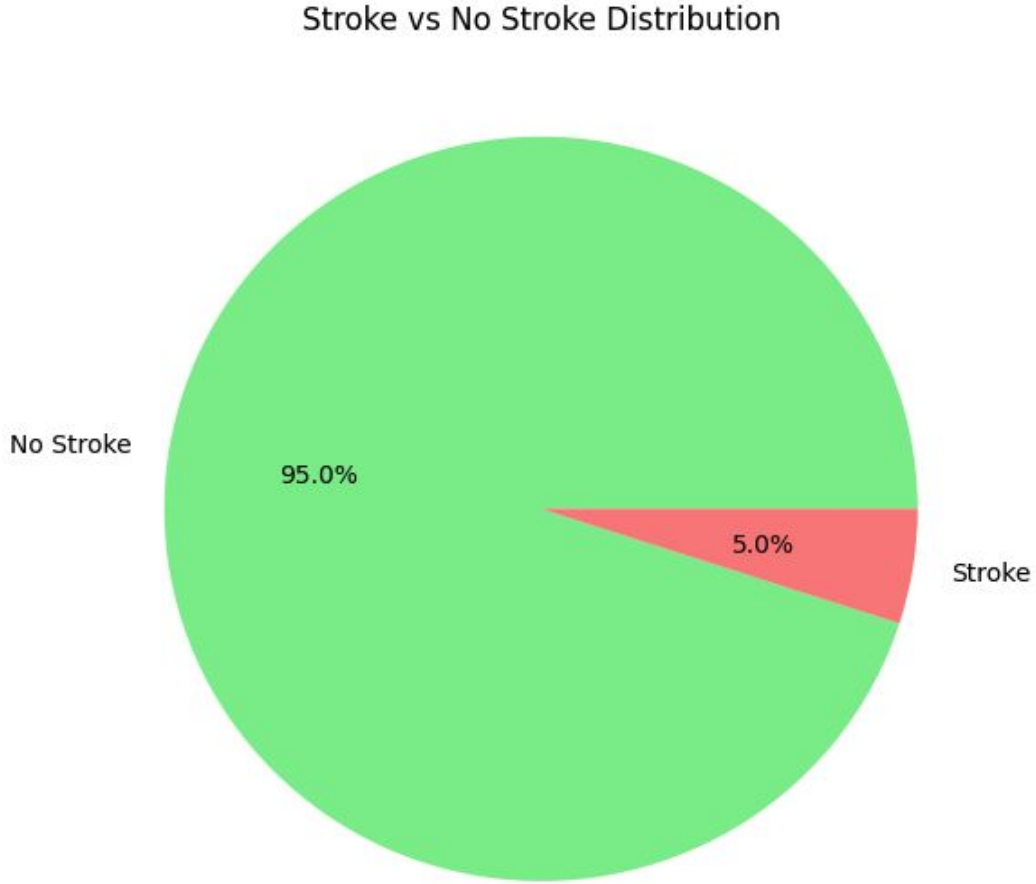
used get_dummies to one-hot encode out categorical values

Split off the target variable and got to work on the first model to begin experimenting

Initial Traditional ML Analysis

Ran a Nearest Neighbors model and a Random Forest model of the data initially and discovered an accuracy score of 95% about non-stroke pts, but for predicting a stroke the accuracy score was a big zero, so we dug back into the data. This obviously threw up some red flags for us so we went back to look at the data to see if we could make it more robust.

The positive stroke data was only 5% of the dataset and it was all organized at the top of the dataset.


Stroke vs No Stroke Distribution
No Stroke
95.0%
5.0%
Stroke

Discovering and resolving data imbalance classification

- Undersampling - Deleting some "majority" data to balance distribution
- Oversampling - Duplicating "minority" data
- Stratification - Keeping original distribution in train test split

Before Undersampling:

Confusion Matrix

	Predicted 0	Predicted 1
Actual 0	1187	4
Actual 1	54	1

Accuracy Score : 0.9534510433386838
Classification Report

	precision	recall	f1-score	support
0	0.96	1.00	0.98	1191
1	0.20	0.02	0.03	55
accuracy			0.95	1246
macro avg	0.58	0.51	0.50	1246
weighted avg	0.92	0.95	0.93	1246

After Undersampling:

Confusion Matrix

	Predicted 0	Predicted 1
Actual 0	47	18
Actual 1	13	46

Accuracy Score : 0.75
Classification Report

	precision	recall	f1-score	support
0	0.78	0.72	0.75	65
1	0.72	0.78	0.75	59
accuracy			0.75	124
macro avg	0.75	0.75	0.75	124
weighted avg	0.75	0.75	0.75	124

K Nearest Neighbors

Trained the data using x_train, y_train

The first time we printed the classification report we had a 95% accuracy because the model was trained on the No stroke data which was 95%. At this point the model could not predict strokes because it was not trained on the 5% that represented strokes.

to tune the model to make it more accurate using Nearest Neighbors we made a loop to test a range of classifiers. We ended up using 1 as classifier.

K Nearest Neighbors tuning

Applied the K Nearest Neighbors Algorithm to the data and got an accuracy of 76.79% after the additional preprocessing.

Nearest Neighbour - 1

```
best_accuracy = 0
best_n_neighbors = 0

# Try different values of n_neighbors from 1 to 20
for n in range(1, 200):
    knn = KNeighborsClassifier(n_neighbors=n)
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)

    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_n_neighbors = n

print(f"Best n_neighbors: {best_n_neighbors}")
print(f"Best accuracy: {best_accuracy}")
```

K Nearest Neighbors Classification Report

	precision	recall	f1-score	support
0	0.81	0.70	0.75	118
1	0.74	0.83	0.78	119
accuracy			0.77	237
macro avg	0.77	0.77	0.77	237
weighted avg	0.77	0.77	0.77	237

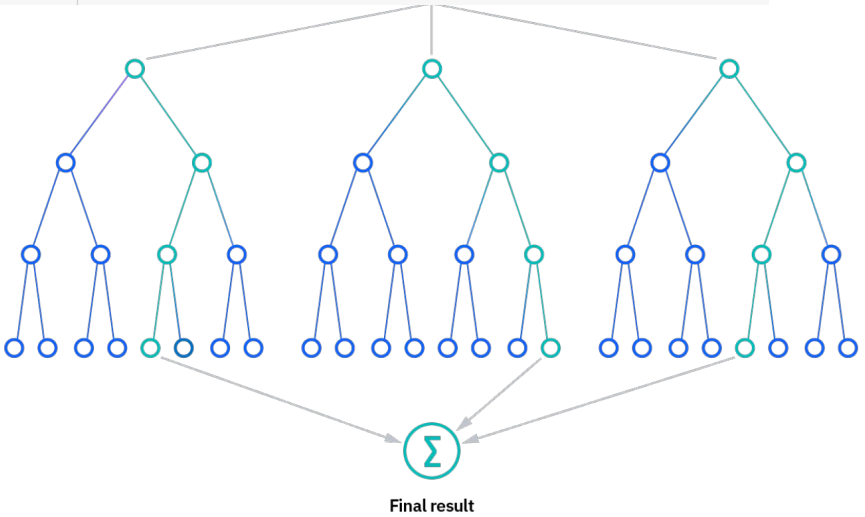
The accuracy is 76.79%

Random Forest Analysis

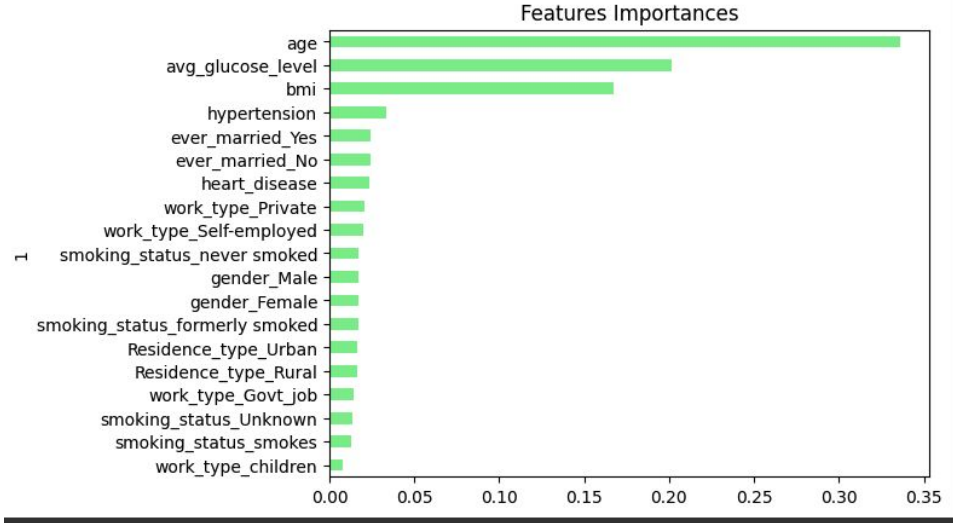
Random Forest Model - Decision Trees Combined

Experimented with changing number of estimators + random_state

```
[14] #init rf model and fit to training data
     rf_model = RandomForestClassifier(n_estimators=3000, random_state=42) #model max acc at n_estimators = 3000, acc loss beginning at 8000
     rf_model = rf_model.fit(X_train, y_train)
```



Important Features - RFM



Confusion Matrix

	Predicted 0	Predicted 1
Actual 0	82	36
Actual 1	17	102

Accuracy Score : 0.7763713080168776

Classification Report

	precision	recall	f1-score	support
0	0.83	0.69	0.76	118
1	0.74	0.86	0.79	119
accuracy			0.78	237
macro avg	0.78	0.78	0.77	237
weighted avg	0.78	0.78	0.77	237

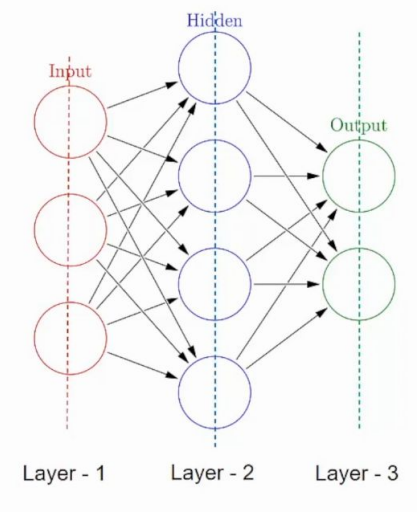
Neural Network Analysis

Auto optimization params plugged in

Added and subtracted hidden layers/number of nodes/number of epochs

Changed optimizer

Ended up with a 75.9 percent accuracy after 50 epochs and NADAM



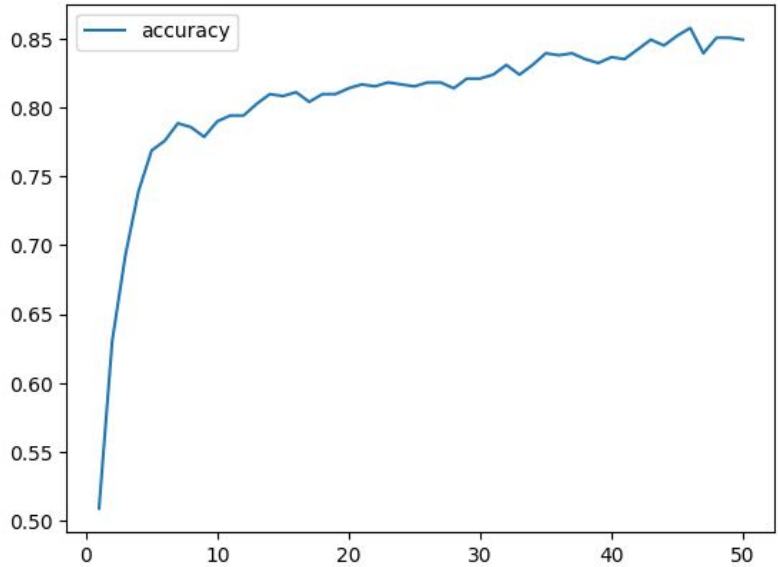
Troubles with relu

After working so long with tanh, we tried changing the activation to relu

That turned our model into a guessing machine

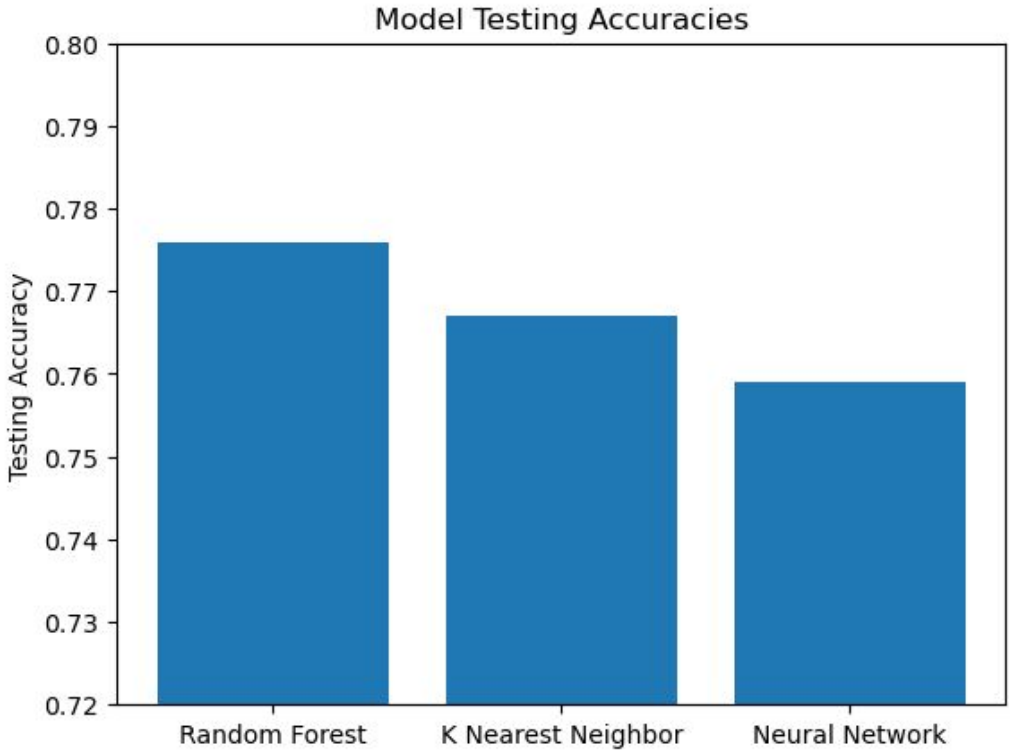
Investigated seeding random states

Neural Network Accuracy




```
8/8 [==============================] - 0s 3ms/step - loss: 0.5718 - accuracy: 0.7595
Loss: 0.5717619061470032, Accuracy: 0.7594936490058899
```

Model Accuracy



recommendations

With a similar level of accuracy AND with a greater level of explainability, our recommendation to this clinic would be to utilize the random forest method of ML to help identify pts at risk of stroke.

Further, an actual clinic would have access to more data than this hackathon data possibly resolving our imbalanced dataset problems. If not a more robust sampling method could improve accuracy.

Our final recommendation would to use a cloud computing service as tensorflow's gpu setup is complicated and WILL break your PC or environment.

questions