

Branch Prediction

Introduction

Modern CPU architectures employ pipelining where instructions can be fetched ahead of time and executed in parallel, removing the need to wait for the termination of the previous instruction. This increases the throughput of the CPU and the performance of the system. However one of the instructions that is heavily used by CPUs pose a challenge to this pipelining architecture: branch instructions. Because the instruction following a branch instruction can be the target of it or the sequential successor of it. The CPU tries to guess the next instruction when it encounters a branch instruction, however if it guesses wrong, a bunch of calculations have to be thrown away and CPU has to start over from the correct instruction. This stalls the CPU and diminishes performance. For this reason various branch prediction techniques have been employed. We wanted to attempt to solve this problem with machine learning techniques.

Data Collection and Feature Extraction

Our raw dataset is a log of CPU instructions that were executed as we played the game Doom on an emulator. We have chosen Doom We specifically logged the address of branch instructions and the address of the instruction that was executed immediately after. This allowed us to determine whether a branch was taken or not by comparing those addresses. If the branch was not taken then the second address would be 2 more than the first, if it was not, it means that the branch was taken.

Our raw data was about several hundreds of megabytes in size, which was collected in the duration of playing the game for several minutes. We then proceeded to extract features from this raw data. There was a study that was done with perceptrons to achieve branch prediction.[1] They used the bits of the global branch history shift register as the feature set of the perceptron model. Seeing as they were successful in doing so, we decided to use the global branch history as the features of our model. For this purpose, we wrote a Python script that went through a portion of

the raw data and recorded the branch history and whether the branch was taken or not. After this, our data was ready to be fed into the model. We had a total of eight features and a binary label. A label of 1 meant the branch was taken and a label of -1 meant the branch was not taken. A feature $x(i)$ meant, whether the $(i+1)$ th branch preceding the current instruction was taken or not. An example of this could be:

features: [1, -1, -1, 1, -1, 1, 1, -1]

label: -1

This means the last branch was not taken, The second last was taken. The third last was taken and so on. And also the current branch was not taken. We used a total of 2000 samples, 1000 1's and 1000 -1's.

# samples	2000
# categories	2
# class 1 in data	1000
# class -1 in data	1000
# dimensions of the feature vector	8

There were some initial problems with our data. Because we first tried to collect data while running another game (Pokemon), however there were long stretches of empty loops that gave the same set of branch and target instructions over and over again. So it didn't provide any variability in the data. For this reason we collected the data from another game (Doom), which gave more data variability. We think this is because Doom is a 3D game which doesn't use GPU and makes all the calculations in the CPU. So it has to make a lot of intensive and high variability calculations. However even after that the problem persisted to some degree. For this reason we actually purged the section of the data that were essentially just busy waits.

After this step, we realized that our dataset was highly skewed towards class 1, and the models that we trained just learned to guess 1 to lazily increase the accuracy metrics. To counter that we artificially balanced the dataset by selecting an equal number of samples from each class to feed into our models.

To measure the interclass and intraclass similarities of our dataset we used two similarity measures: Euclidean distance and cosine similarity. You can see the similarity scores below.

Cosine Similarity	Euclidean Distance
Interclass = -0.032	Interclass = 3.96
Intraclass (-1) = 0.084	Intraclass (-1) = 3.66
Intraclass (1) = 0.748	Intraclass (1) = 1.01

As we can see from the similarity scores above, intraclass similarity of class -1 is higher than that for class 1, indicating more variability. Interclass similarities are relatively low.

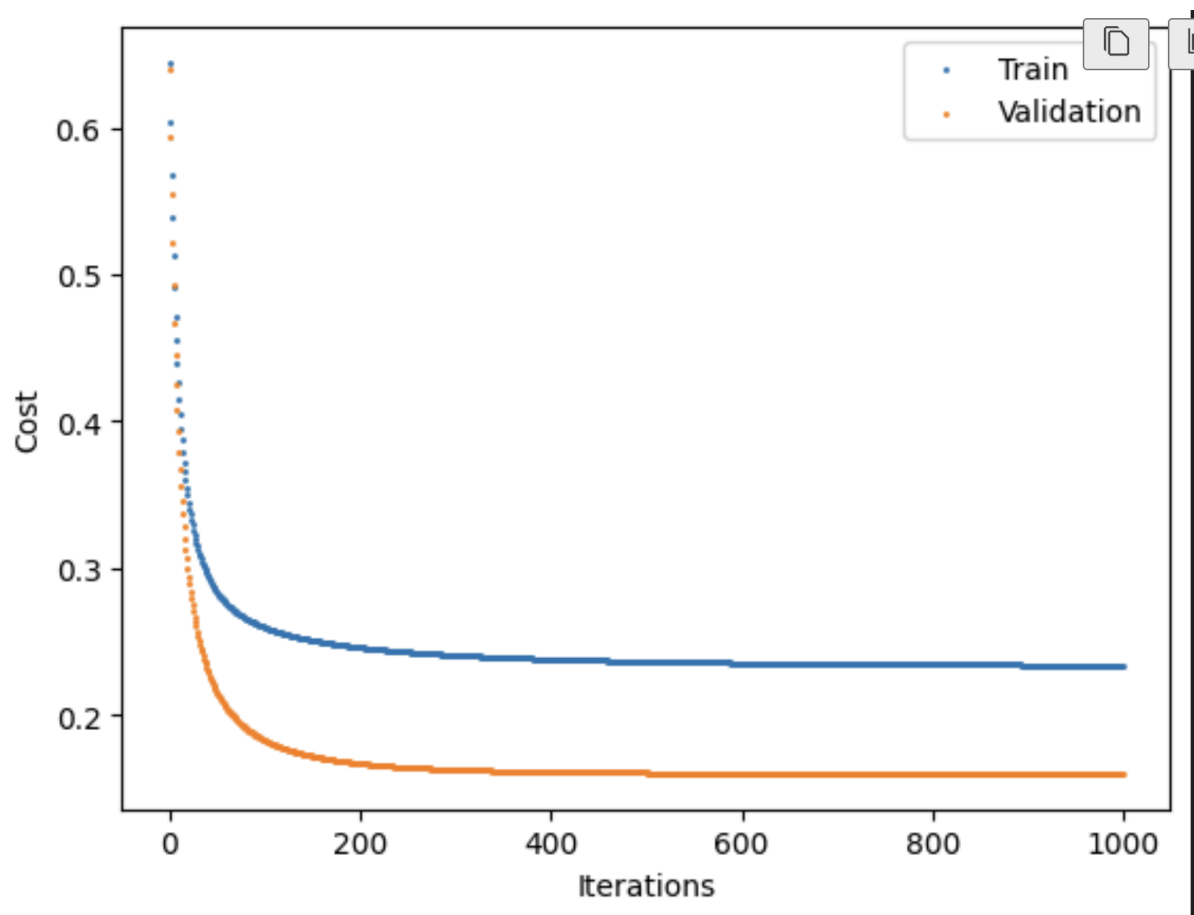
To evaluate the difficulty of our dataset, we used the k-means algorithm that we implemented from scratch and clustered our dataset into 2 groups and plotted the results. There wasn't a super clear linear boundary however it is still linearly separable, which means our dataset isn't superhard, in the sense that it won't require non-linear techniques.



Supervised Learning

A) Logistic Regression

We implemented logistic regression from scratch and trained a model on our dataset. To see if there is overfitting, we plotted the loss over time as it can be seen below.



If there was overfitting, we would see validation loss increasing while training loss got better. However training loss and validation loss kept getting better and stopped improving together, which suggests that there was no overfitting. So we didn't implement regularization.

Our stopping criteria for logistic regression is if the validation loss doesn't improve for a certain number of iterations. We set that number to 10, however the training didn't make an early stop and went for the maximum iteration count that we set.

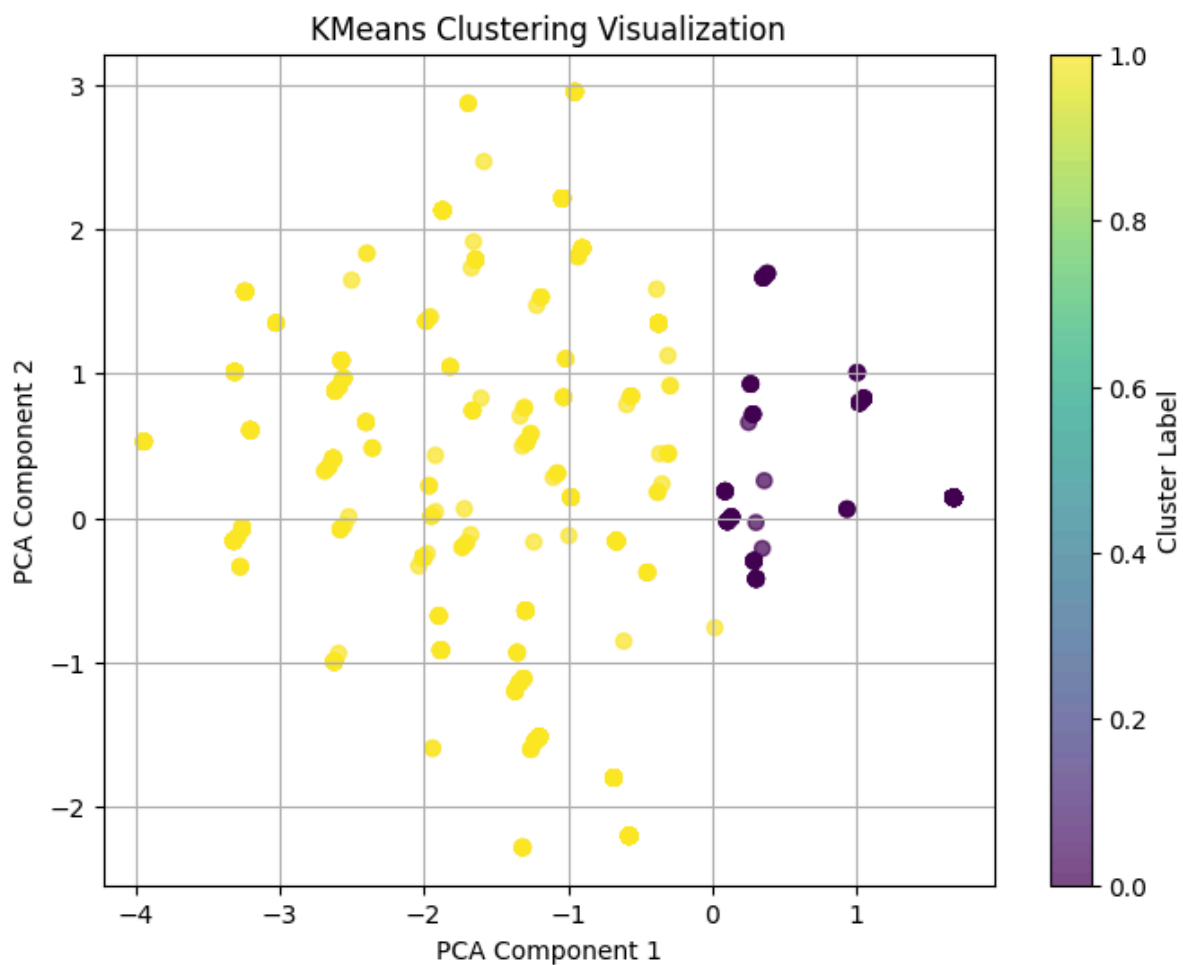
Our training accuracy is %93 percent and test accuracy is %91.75.

The Scikit-learn model gives an accuracy of %93 percent.

Our function takes 0.48654199 seconds to train.

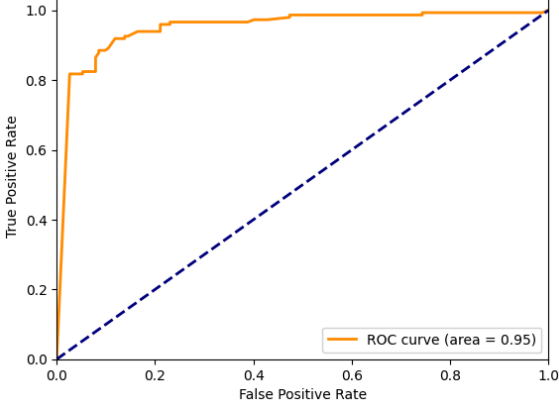
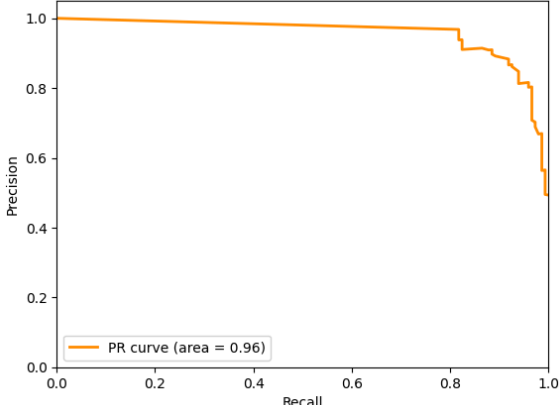
Scikit learn takes 0.00004506 seconds to train.

Our data after being clustered by our custom k-means implementation looks like this.



Those clustering results suggest that members of class 0 occupy a smaller space in our vector space. They are more densely located and hence more similar to each other.

Performance Metrics:

ACC	0.8933333333333333
AUROC	<div><div>Receiver Operating Characteristic (ROC)</div><p>ROC curve (area = 0.95)</p></div>
Precision-Recall	<div><div>Precision-Recall Curve</div><p>PR curve (area = 0.96)</p></div>
F1-Score	0.8933333333333333

B) Support Vector Machine

Firstly, we have implemented a linear soft-margin support vector machine from scratch. For this, we have used Cvxopt library's quadratic program solver. Below are explanations of what we have fed to the solver:

1-) P : The $N \times N$ matrix (N : number of data points) whose (i, j) th component is $y_i y_j \langle x_i, x_j \rangle$, where $\langle *, * \rangle$ is our inner product (in our scenario, this is basic dot product) and x_i and x_j are i th and j th data vectors. This will correspond to the summation of $\frac{1}{2} (a_i a_j y_i y_j \langle x_i, x_j \rangle)$ values as i and j ranges from 1 to n .

2-) q : The N dimensional vector with all its values filled with -1. This will correspond to the summation of a_i values where i ranges from 1 to n .

3-) G : Two vertically stacked $N \times N$ matrices with diagonal values -1 and 1 respectively. Corresponds to our constraint $0 \leq a_i \leq C$ as i ranges from 1 to n .

4-) h : Two horizontally stacked N dimensional vectors which have all its values 0 and C respectively. Corresponds to our lower and upper bound for our a vector (whose i th component is a_i), which are 0 and C respectively

5-) A : The $1 \times N$ matrix created with reshaping the N dimensional vector y (vector of our data points' labels). Corresponds to (summation) $a_i y_i$

6-) The 1×1 matrix with its only value as 0. 5-) and 6-) together correspond to the equality constraint (summation) $a_i y_i = 0$

We first applied a 5-fold cross validation with our training data (dividing %80 of our entire training data as %64-%16 for each of the 5 folds) with candidate C values as [0.01, 0.1, 1, 10]. Our average accuracy results for each value is as follows:

C value: 0.01, Average Accuracy: 0.9081

C value: 0.1, Average Accuracy: 0.9200

C value: 1, Average Accuracy: 0.9144

C value: 10, Average Accuracy: 0.9163

According to the above results, the best C value is chosen as 0.1. Here are our test results for different kernels using %80 train %20 test data. Note that here we are using Sklearn library SVM, not the scratch model:

Kernel: linear, Accuracy: 0.9175

Kernel: rbf, Accuracy: 0.9075

Kernel: poly, Accuracy: 0.9375

Kernel: sigmoid, Accuracy: 0.8900

In order to compare our scratch SVM model and Sklearn SVM model, here are their test accuracies and runtimes:

Scratch Model Test Accuracy: 91.75%

Scratch SVM Training Runtime: 6.3054 seconds

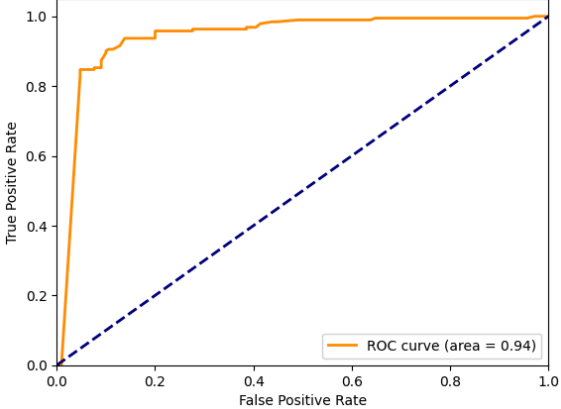
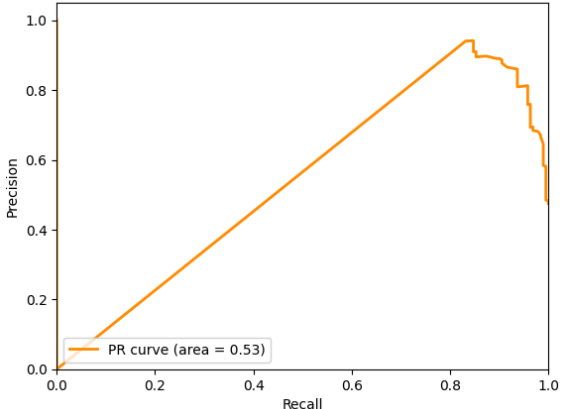
Scratch SVM Testing Runtime: 0.0004 seconds

Scikit-learn Model Test Accuracy: 91.75%

Scikit-learn SVM Training Runtime: 0.0656 seconds

Scikit-learn SVM Testing Runtime: 0.0144 seconds

Performance Metrics:

ACC	0.9
AUROC	<div><div>Receiver Operating Characteristic (ROC)</div><p>ROC curve (area = 0.94)</p></div>
Precision-Recall	<div><div>Precision-Recall Curve</div><p>PR curve (area = 0.53)</p></div>
F1-Score	0.9000400641025641

C)Random Forest

In this part, we have directly used Sklearn library's Random Forest Classifier function. We first split the dataset %80-%20 as train and test datasets respectively. Then we applied 5-fold cross validation (%64-%16 for each of 5 folds) to our training set (note that in this setting, the %16 part somehow works as the "validation dataset" as the whole %80 train dataset will indeed be used for evaluating the final version of our model) for deciding the number of trees with candidate values [5, 10, 50, 100, 200] respectively. Here are the results:

Number of trees: 5, Average Accuracy: 0.9500

Number of trees: 10, Average Accuracy: 0.9481

Number of trees: 50, Average Accuracy: 0.9519

Number of trees: 100, Average Accuracy: 0.9500

Number of trees: 200, Average Accuracy: 0.9519

From the results above, we decided to choose the number of trees as 50 as it gives the highest average accuracy on 5-fold cross validation. (it has an equal average accuracy with 200 trees, however 50 tree model is simpler than the 200 tree one, so we chose that model) Now putting our train and test datasets onto our Random Forest model with 50 trees, we obtain the results below:

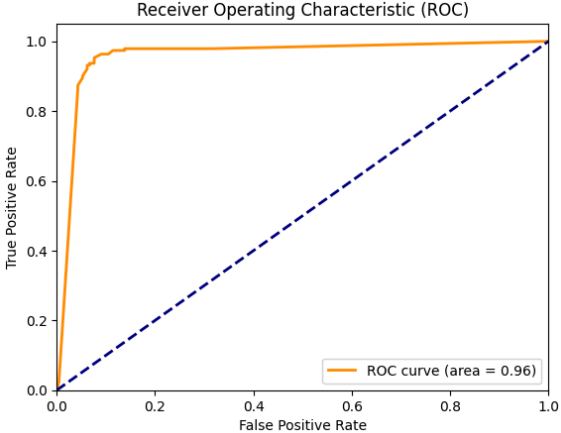
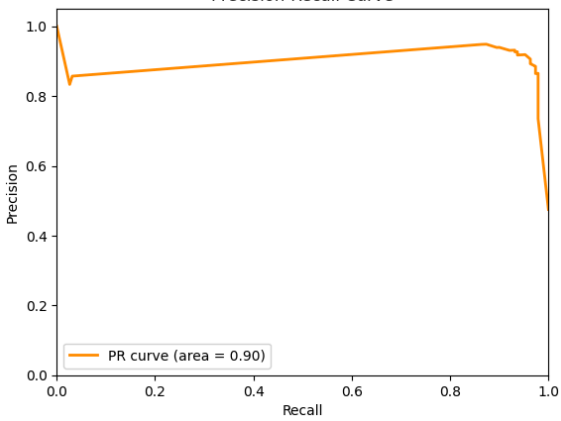
Accuracy: 0.9400

Random Forest Training Runtime: 0.1005 seconds

Random Forest Evaluation Runtime: 0.0091 seconds

Random Forest Prediction Runtime: 0.0076 seconds

Performance Metrics:

ACC	0.9325
AUROC	<div><p>Receiver Operating Characteristic (ROC)</p><p>ROC curve (area = 0.96)</p></div>
Precision-Recall	<div><p>Precision-Recall Curve</p><p>PR curve (area = 0.90)</p></div>
F1-Score	0.9325080337511511

D)k-Nearest Neighbors (KNN)

In this part, we have directly implemented the KNN algorithm from scratch. For that part, we first defined some possible distance metric functions inside `distance_function()` function, and then used it inside our KNN algorithm. In the algorithm part, we yet again split the dataset %80-%20 as train and test datasets, and applied 5-fold cross validation (%64-%16 for each of 5 folds) to our training set for deciding the value of K with candidate values [1,3,5,7,9] respectively. Here are the results:

K: 1, Average Accuracy: 0.9425

K: 3, Average Accuracy: 0.9437

K: 5, Average Accuracy: 0.9450

K: 7, Average Accuracy: 0.9469

K: 9, Average Accuracy: 0.9456

From the results above, we decided to choose our value K as 7 as it gives highest average accuracy on 5-fold cross validation. Now putting our train and test data onto our Random Forest model with K=7 with different distance metrics, we obtain the results below:

Distance Metric: euclidean, K: 7, KNN Accuracy: 93.75%

KNN Prediction Runtime: 0.0635 seconds

Additionally, we have tried using different distance metrics with our train and test datasets with our obtained best K value (K=7). Here are the results:

Distance Metric: euclidean, K: 7, KNN Accuracy: 93.75%

Distance Metric: cosine, K: 7, KNN Accuracy: 94.00%

Distance Metric: manhattan, K: 7, KNN Accuracy: 93.75%

Distance Metric: gaussian, K: 7, KNN Accuracy: 36.25%

Distance Metric: L_p, K: 7, KNN Accuracy: 93.75%

Distance Metric: chebyshev, K: 7, KNN Accuracy: 91.25%

Note: In Gaussian metric, sigma is chosen as 1; in L_p metric, p is chosen as 2

Performance Metrics:

ACC	0.9425
AUROC	0.9430
Precision-Recall	0.9477
F1-Score	0.9412

(We were not able to plot the ROC or PR curve, since we used the average as the final value.)

Runtime Comparison

Logistic Regression	0.0002 secs / 2.24 secs
SVM	0.001 secs / 5.93 secs
Random Forest	0.02 secs / (no scratch time)
kNN	(no library time) / 0.05 secs (prediction time for whole dataset)

Fastest model seems to be logistic regression, most probably due to its simplistic nature.

1. **ACC:** Measures the overall correctness of the model. It is a great metric to use for balanced datasets. Since all the dataset contributes to this metric, imbalances in the dataset would diminish in importance for classes with less data.
2. **AUROC:** Measures how well the model ranks positive samples higher than negative samples. If we do not know the threshold, this metric would help us. Even though it can be useful for imbalanced datasets, it does its job best when the problem is a balanced binary classification problem.
3. **Precision-Recall:** These are great metrics for detecting positive instances. Precision checks at confidence in the positive prediction and recall checks for

detecting as many positives as possible. Since the number of data for classes is not important in this metric, it is great for imbalanced datasets.

4. **F1-Score:** It is used for balancing precision and recall. Thus, it is great to use when we care about both FPs and FNs. Since it is calculated from precision & recall, which does not care about imbalance in the dataset, F1-score is a great metric to use when the data is imbalanced.

Random forest has the best performance in terms of evaluation metrics. We think it's because the random forest has a better chance of capturing even non-linear features without assuming an underlying data distribution.

Challenges We Faced

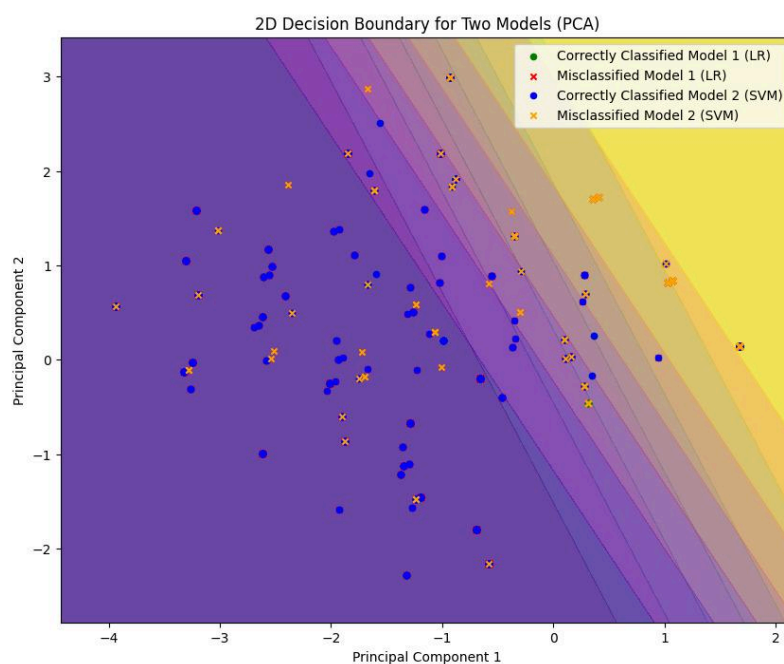
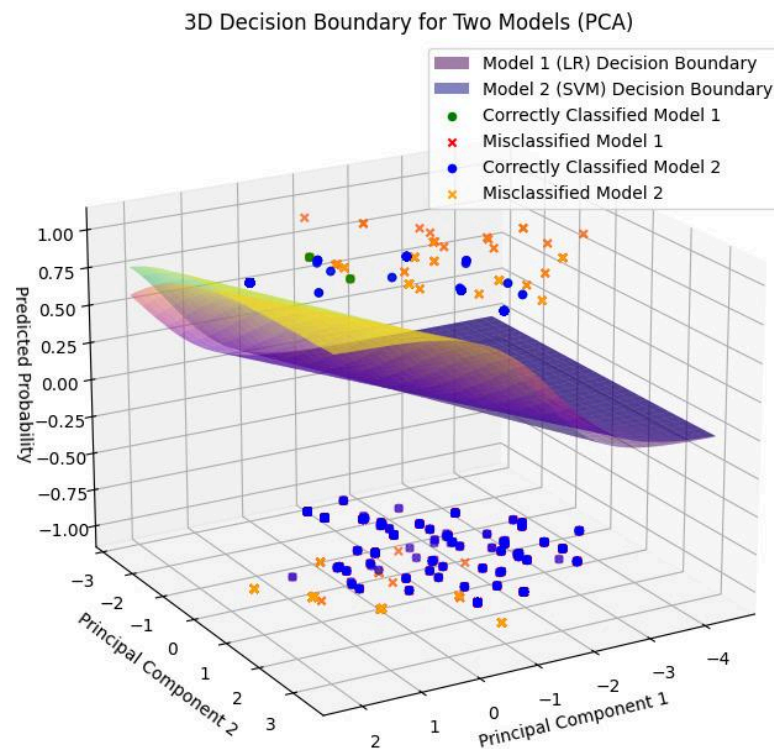
Getting the data: Since we needed to get results for branch instructions in a CPU, we had to simulate the CPU over an emulator or FPGA. We had chosen an emulator for ease of use. We found an open source emulator for an ARM7chip and we injected a branch logger code. We were challenged with slow emulation because of the logger. We fixed it by buffering the logger which caused compiler errors but we were able to fix it.

Extracting the features: The logger has given us hundreds of megabytes of data in just minutes and most of it was garbage data which was generated by busy wait functions in the emulated code. We solved these issues by changing the code with a more computation intensive code (Pokemon to DOOM) and filtering the data which has taken a branch 3 times in a row in the local history(For the same branch instruction).

Training the models: During the training phase of Logistic Regression (Sklearn), SVM, Random Forest, KNN and K-means; we did not have such issues as our datasets are already split well as train and test models in a randomized manner, and the model implementations are either working seamlessly or already library models. However at the Logistic Regression's scratch part, we had some issues. At first, our model was implemented wrong, so both our train and test accuracies were too low. Later, we implemented the correct scratch model for logistic regression and it was running well with good train and test accuracies. Luckily our model did not overfit, so we did not need to implement regularized logistic regression as well.

Plotting Decision Boundary

To plot the decision boundaries of our models, we followed two different strategies. First one is selecting two features from our feature set and plotting a decision boundary on them. Second one is using principal component analysis to select two new axes and plot a decision boundary on them. You can see those two decision boundaries along with the training dataset below.



Decision boundaries and accuracies are pretty similar overall.

Resources

1. <https://www.cs.utexas.edu/~lin/papers/hpca01.pdf>

Group Members and Individual Contribution Statements

Yusuf Kağan Çiçekdağ

- Implemented Logistic Regression (library), SVM (scratch + library), Random Forest and KNN models
- Tested and optimized our models according to our project description
- Fixed the bugs inside our codes
- Contributed to the report, particularly at the parts about the models I have implemented

Ömer Faruk Erzurumluoğlu

- Collected the data by injecting a logger inside the emulator we have chosen.
- Contributed to feature extraction and filtering of the data.
- Generated the plots and calculated the performance metrics.
- Contributed to the report.

Özgür Deniz Demir

- Extract features from raw data.
- Implement logistic regression
- Implement k-means
- Apply PCA and plot decision boundaries
- Evaluate models on certain metrics and plot loss curve over time
- Wrote the introduction section, logistic regression and decision boundary part of the report, added relevant metrics and plots.

Link to our zip file:

https://drive.google.com/drive/folders/1ZjHeTPBa_VknwbKGLNVJ9w4Ilhaq7FEH