Pokémon Image Recognition

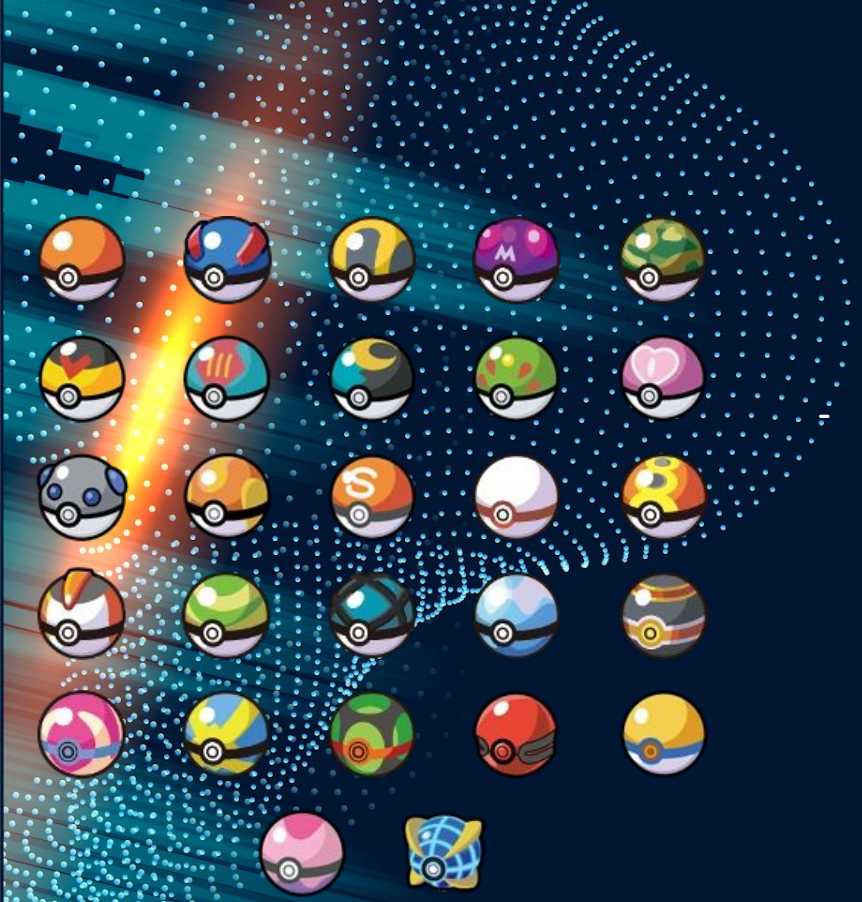
CS 5232 - Machine Learning and Data Mining

Group 3: Eric Nieters, Ritwik Raj Saxena, and Ibrahim Mamudu

Outline

- Introduction
- Proposed Work
  - Implementations of algorithms
  - Datasets
- Code Overview
- Live Demo
- Results
- Conclusion

Source: https://pokecoord.com/pokemon-nests/



Source: https://nintendo.fandom.com/wiki/Poké_Ball

Introduction

<u>The Problem:</u>

- Pokémon image recognition is uncommon.
- Existing models detect Pokémon, but provide no other information.
- Most programs recognize few Pokémon.

<u>The Solution:</u>

- Implement a website for detecting Pokémon.
  - Provides easier access than software.
  - Detect ⅔ of all currently existing Pokémon.

Introduction (Part 2)

Our algorithms (most accurate to least accurate):

1. Convolutional Neural Networks
2. Pre-trained Models
3. Support Vector Machines
4. K-Nearest Neighbor

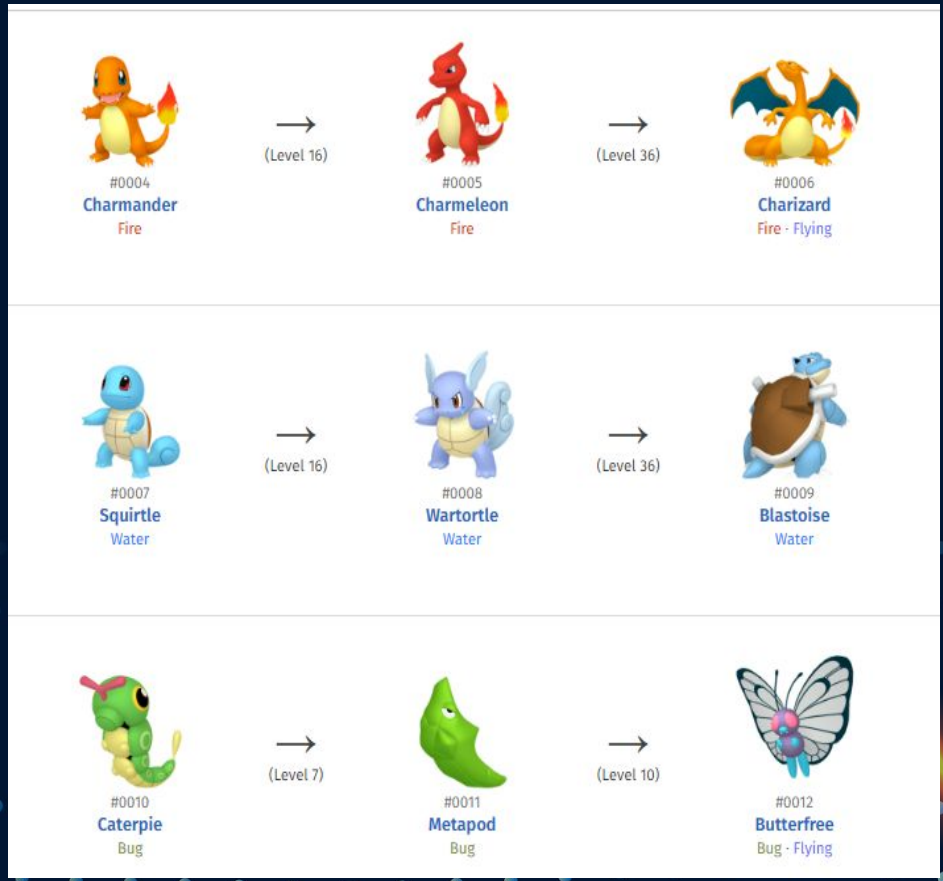
Other Features:

- Website implementation
- Compare your image to the predicted Pokémon
- Display stats of the predicted Pokémon
- Analyze all of the stats of the predicted Pokémon

Source:
https://projectpokemon.org/home/gallery/image/78446-charmanderpng/

An Introduction to Pokemon

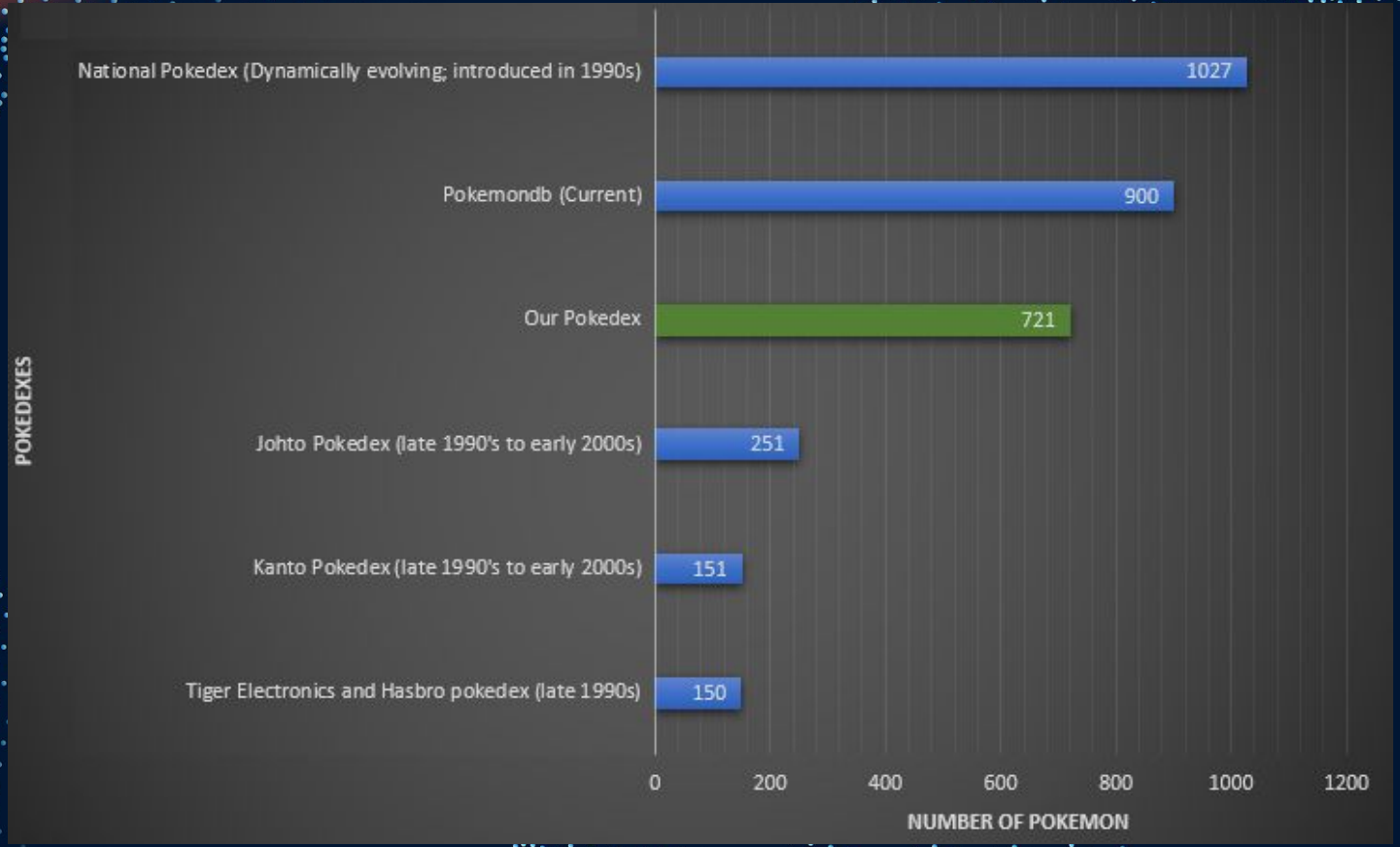
A pokemon is a fictional creature which is the basis for certain Japanese and Japan-inspired games and cartoons. It has stats, a type, a generation, and an evolution paradigm, as shown in the figure on the right. A list of the categories under which a pokemon can fall is as under:



Proposed Work

- Supervised learning was used to train models.
  - Read in 10-20 images of each of 721 Pokémon.
  - Collect data, such as patterns, colors, number of pixels or size, depending on the model.
  - Store the model's data and load it whenever a prediction is being made.
- Developed in Python. Website uses PHP, HTML, CSS, JavaScript, jQuery, and the Bootstrap framework.
- All models are pre-trained. This reduces website load.

Comparison of Pokédexes



Convolutional Neural Networks (CNN)

1. Each layer performs convolutions on images.
2. Convolutions enable the architecture to extract features from each image exhaustively.
3. CNNs are usually deep to very deep architectures, depending on the number of convolution operations involved. Usually higher the number of convolutions, better is the feature extraction, and better is the prediction power of the model.
4. They are usually used as the pretrained models as well as the new classifiers in a Transfer Learning setup for object or image classification.

Distinctive Features of Convolutional Neural Networks (CNN)

We used Convolutional Neural Networks for image classification because

1. **Pattern identification**: They were created specifically for image classification; they efficiently recognize patterns from the images.
2. **Color recognition**: They focus on efficacious color recognition and different color planes
3. **Critical feature isolation**: They minutely track extremely critical features, like edges, on each image to produce an immensely precise prediction.

Transfer Learning with Pre-Trained Models

- Our transfer learning utilizes the library called **`SentenceTransformers`**
  - A library for sentence, text and image embeddings.
  - Similar to "implementing Google" using your own detected vocabulary.
  - Works with multiple languages.
- Detects over 1,000,000,000+ pairs by using a 256 dimension vector field that can plot images and sentences near each other.
  - K-Nearest Neighbor on steroids.
- Can detect the top five most similar pairs of sentences:

```
sentences = ['A man is eating food.',
             'A man is eating a piece of bread.',
             'The girl is carrying a baby.',
             'A man is riding a horse.',
             'A woman is playing violin.',
             'Two men pushed carts through the woods.',
             'A man is riding a white horse on an enclosed ground.',
             'A monkey is playing drums.',
             'Someone in a gorilla costume is playing a set of drums.'
             ]
```

Source: https://www.sbert.net/examples/applications/image-search/README.html

K-Nearest Neighbor

- Our “from scratch” algorithm which uses no machine learning libraries.
- All images of Pokémon get converted into an array that counts the number of times that 256 total colors appear in each image.
- Euclidean Distance is measured between the tested Pokémon and every single Pokémon image in the dataset.
    - The “shortest” distance between the trained set and the tested Pokémon is stored in ranking order.
- The most difficult algorithm to implement for our project!

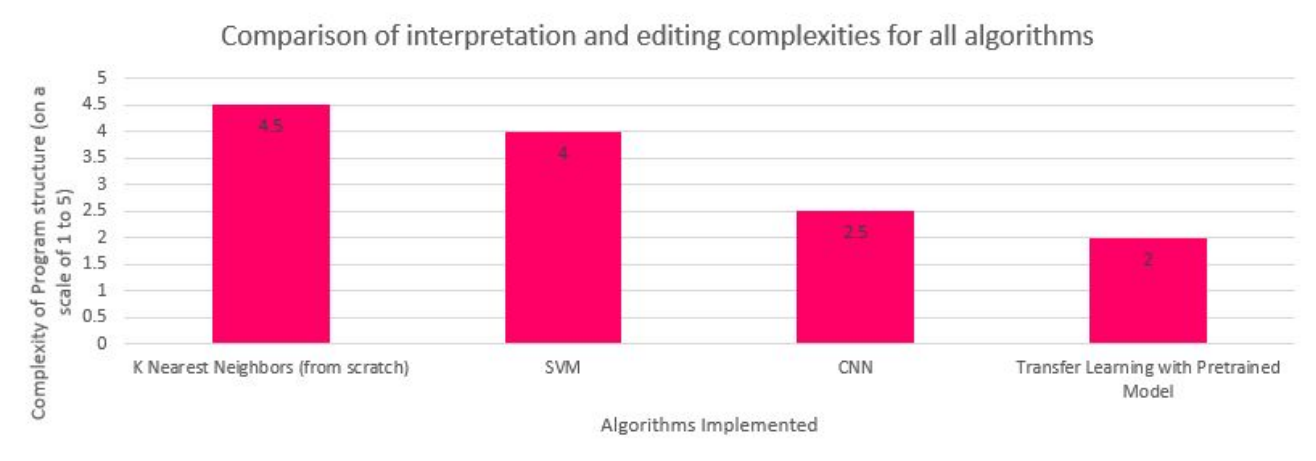
Source: https://pngimg.com/image/27696

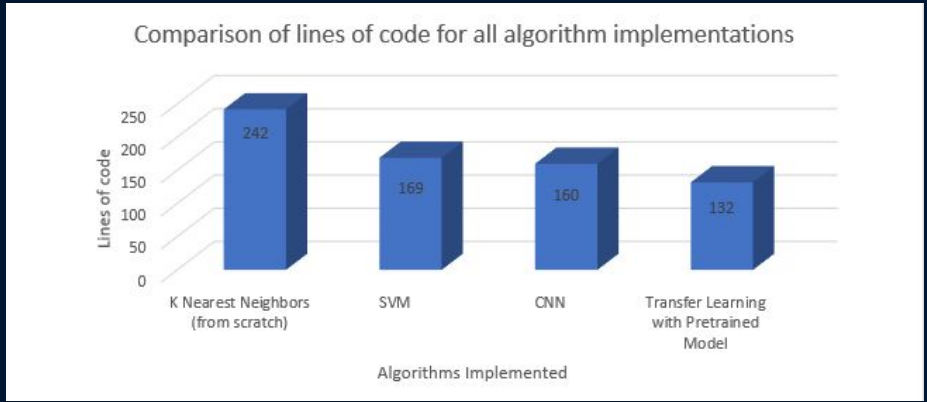
Support Vector Machines (SVM)

- SVMs aim to find the optimal hyperplane to separate different classes while maximizing the margin between the closest data points (support vectors).
- Not very well suited to image recognition, but work well for smaller datasets and “plottable” data.
- Was still utilized in the experiment to show the which algorithms must and must not be used for image detection.
- Accuracy was difficult to improve upon, as well as computation time.
    - The least recommended algorithm for a project like this.

Source: https://www.stickpng.com/img/games/pokemon/bulbasaur-pokemon

Comparison of Intricacy of our Algorithms





Datasets

Image data:

- Generated by us.
- Originally contained 14,420 images.
- 4,347 images removed by hand that were bad data, leaving 10,073 images in our final dataset.
  - The images that were removed had detailed background, props, multiple Pokémon in one image, etc.
- Used `BingImageDownloader` in Python.

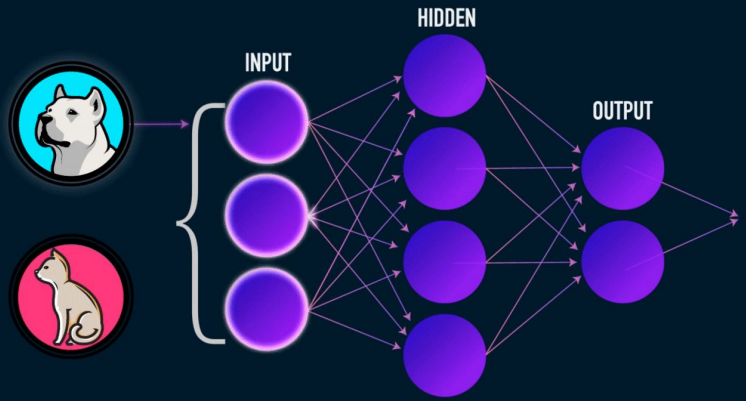
Pokémon statistical data:

- Generated by alopez247 on Kaggle.
- Contains 721 Pokémon from generations 1 through 6.
- Each entity has 21 attributes.
  - Details range from stats from the games to the type of egg that it's hatched from.
- 21 years worth of information.
- All displayed on the website.

Source:
https://www.kaggle.com/datasets/alopez247/pokemon

Code Overview (CNN)

- InceptionResNetV2 was used from the Keras/Tensorflow Python library.
- A pretrained model trained on Imagenet dataset.
- Uses the Inception family of architectures, created by Google.
- Integrates Inception model with the idea of residual connections (ResNet).
- Described as a very deep architecture.
- Was developed with the aim to enhance algorithmic performance, primarily in image detection, classification, processing and prediction.



Our InceptionResNetV2 Model uses four layers:

```
# Build Inception-ResNet model
base_model = InceptionResNetV2(include_top=False, weights='imagenet', input_shape=(img_height, img_width, 3))
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(1024, activation='relu')(x)
predictions = Dense(num_classes, activation='softmax')(x)
```

1. InceptionResNetV2 Base Model:
   a. InceptionResNetV2 is a deep CNN architecture combining the Inception architecture with residual connections.
   b. This first line of the code creates the base model using InceptionResNetV2 with the following settings:
      i. `include_top=False`: Excludes the fully connected layers at the top of the network, allowing for custom additional layers.
      ii. weights='`imagenet`': Initializes the model with weights pre-trained on the ImageNet dataset.
2. Global Average Pooling Layer: `GlobalAveragePooling2D()` layer reduces each feature map to a single value by averaging all values in the feature map, providing a fixed-size output.
3. Dense Layer (with ReLU Activation):
   a. `Dense(1024, activation='relu')` :Adds a fully connected layer with 1024 units and ReLU activation function.
   b. This layer performs a linear transformation on the incoming data followed by an element-wise ReLU activation function.
   c. It introduces non-linearity (through ReLU) to the network and helps learn complex patterns in the data.
4. Output Dense Layer (with Softmax Activation):
   a. `Dense(num_classes, activation='softmax')`: Adds the final dense layer with `num_classes` units (representing the number of classes in the classification task) and a softmax activation function.
   b. Softmax activation calculates class probabilities for multi-class classification.

Code Overview (Pre-Trained)

```python
def run_pretrained(user_input):
    # This model is used specifically for vectorizing images.
    knn_model_trainer = SentenceTransformer('clip-ViT-B-32')
```

```python
# And compute the embeddings for these images
img_emb = knn_model_trainer.encode(img_names)

# And compute the text embeddings for these labels
text_emb = knn_model_trainer.encode(labels)

# Now, we compute the cosine similarity between the images and the labels
cos_scores = util.cos_sim(img_emb, text_emb)

# Then we look which label has the highest cosine similarity with the given images
pred_labels = torch.argmax(cos_scores, dim=1)

# Preparing the variables for statistics:
predicted_correctly = 0
overall_guessed = 0

for img_name, pred_label, correct_labels in zip(img_names, pred_labels, correct_labels):
    # if (user_input == 1):
    #     display(Image(img_name, width=200))
    # print("Predicted label:", labels[pred_label])
    # print("Correct label: " + correct_labels)
    # print("\n\n")

    if (labels[pred_label] == correct_labels):
        predicted_correctly += 1

    overall_guessed += 1

return ([(predicted_correctly / overall_guessed), (time.process_time() - start)])
```

Code Overview (K-NN) - Part 1

```python
# Generates the Euclidean Distance between two vectors. These vectors are 256 in length (for the pixels).
def EucDisHistograms(H1,H2):
    distance = 0
    for i in range(len(H1)):
        distance += np.square(H1[i]-H2[i])
    return np.sqrt(distance)

def generate_training_data_with_export(export=True):
    # Import all the data as a 3D array based on colors:
    training_labels = []
    training_data = []

    # Get all of the images in the training set, then convert them to a 3D vector based off of color, then flatten them and add them to an array.
    # Create another list of the labels for each image, then zip them together so that we have parallel arrays.
    for root, dirs, files in os.walk("../../Project Data/Small Test Set", topdown=False):
        for name in dirs:
            for filename in os.listdir(os.path.join(root, name)):
                # Get the label of the Pokemon:
                training_labels.append(name)
                # Open the image and convert to an array:
                image = np.asarray(Image.open(os.path.join(root, name, filename)))
                # Flatten the image to turn it into a single dimension:
                training_value = image.flatten()
                # Create a histogram where we get the number of times that a pixel appears in the images:
                RH1 = Counter(training_value)

                # Get the counts of all the pixels, put it into the array, then append it to the training data.
                H1 = []
                for i in range(256):
                    if i in RH1.keys():
                        H1.append(RH1[i])
                    else:
                        H1.append(0)

                training_data.append(H1)

    # zipped_array = zip(training_labels, training_data)
    # print(tuple(zipped_array))

    # for i in range(0, len(training_labels)):
    #     print(training_labels[i] + "," + str(training_data[i]) + ",\n")

    if export:
        file = open("knn-data.txt", "w")
        for i in range(len(training_labels)):
            file.write(training_labels[i] + "," + str(training_data[i]) + "\n")

        file.close()
```

Code Overview (K-NN) - Part 2

```
# Get the number of elements that are in the set:
numOfTestSamples = len(training_data)

# Zip the two arrays together so that they stay linked. Cast to list. Then, randomize them for testing.
training_set = list(zip(training_labels, training_data))
random.shuffle(training_set)

# Take a fourth of the list and split the list into two separate lists: Assign testing_set first so that training_set can be mutated:
# If an error occurs here, then you need a bigger test set.
testing_set = training_set[:numOfTestSamples // 4]
# training_set = training_set[numOfTestSamples // 4:]

# For computing the accuracy:
numerator = 0
denominator = len(testing_set)

# For each of the elements in the testing set...
for tests in testing_set:
    # Create an array that we will store the k-number of closest Euclidean distance(s).
    nearestNeighbor = []
    # Go through all the elements in the training set and compute the Euclidean Distance between the test/train sets:
    for trains in training_set:
        result = EucDisHistograms(tests[1], trains[1])
        # print("Nearest Neighbor: " + str(nearestNeighbor))
        if (len(nearestNeighbor) < k):
            nearestNeighbor.append([str(trains[0]), str(result)])

        else:
            for i in range(0, len(nearestNeighbor)):
                # print("Comparing " + str(nearestNeighbor[i][1]) + " with " + str(result))
                if ((float(nearestNeighbor[i][1])) > float(result)):
                    # Delete the value and replace it with the new result.
                    # print("Replacing!")
                    nearestNeighbor[i] = [str(trains[0]), str(result)]
                    # No need to traverse through the rest of the NN array. Go to the next element.
                    break

    # Nearest Neighbor is done. Calculate the best-approximated label:
    bestLabel = []
    for elements in nearestNeighbor:
        bestLabel.append(elements[0])

    # Create a "Counter" dictionary and get the first element, which is the most number of occurrences in the dictionary.
    prediction = next(iter(Counter(bestLabel)))

    # print("\nPrediction: " + prediction)
    # print("Actual answer: " + str(tests[0]) + "\n")
```

Code Overview (SVM)

```
for root, dirs, files in os.walk(dir, topdown=False):
    for name in dirs:
        categories.append(name)
        path = os.path.join(dir, name)
        label = categories.index(name)

        for img in os.listdir(path):
            imgpath = os.path.join(path,img)
            pokemon_image = cv2.imread(imgpath,1)
            pokemon_image = cv2.resize(pokemon_image, (50, 50))
            image = np.array(pokemon_image).flatten()
            data.append([image, label])
```

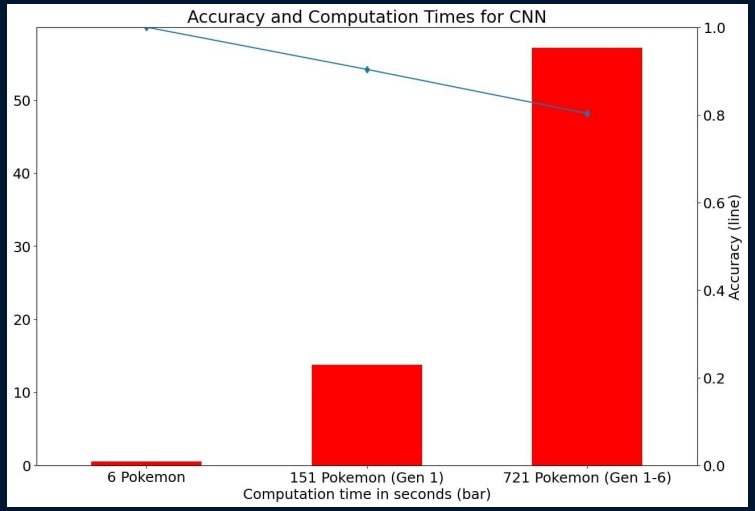
```
random.shuffle(data)
features = []
labels = []

print(data)

for feature, label in data:
    features.append(feature)
    labels.append(label)

print(features)

xtrain, xtest, ytrain, ytest = train_test_split(features, labels, test_size=0.25)

model = SVC(C=1, kernel='poly', gamma='auto')
model.fit(xtrain, ytrain)

with open('svc-model.pkl','wb') as f:
    pickle.dump(model,f)

# prediction = model.predict(xtest)
accuracy = model.score(xtest, ytest)

print("Accuracy: ", accuracy)
# print("Prediction is: ", categories[prediction[0]])
```

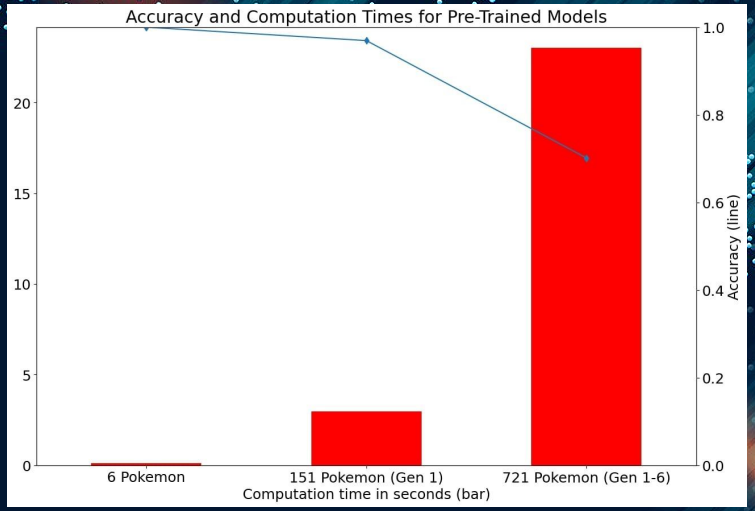
Demonstration

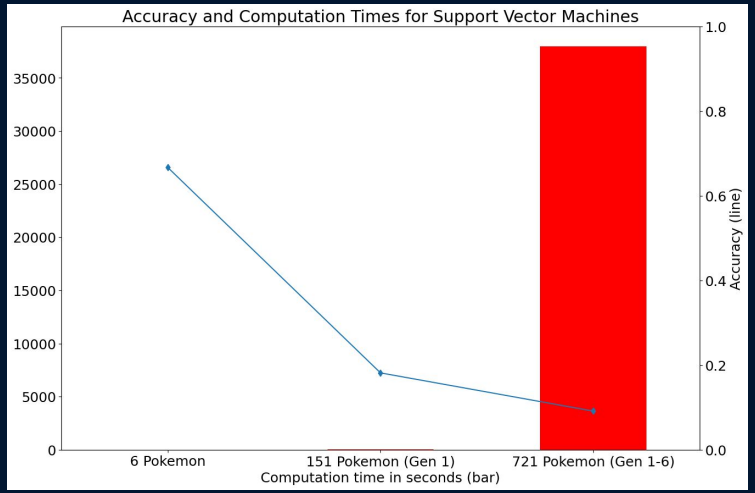
- Due to restrictions from online web hosts, we will be demonstrating on a local web server.
- The website can be viewed on www.pokemonimagerecognition.com. However, the live domain is not functional due to RAM and CPU restrictions.
  - Can also be found by Googling “pokemon image recognition”.

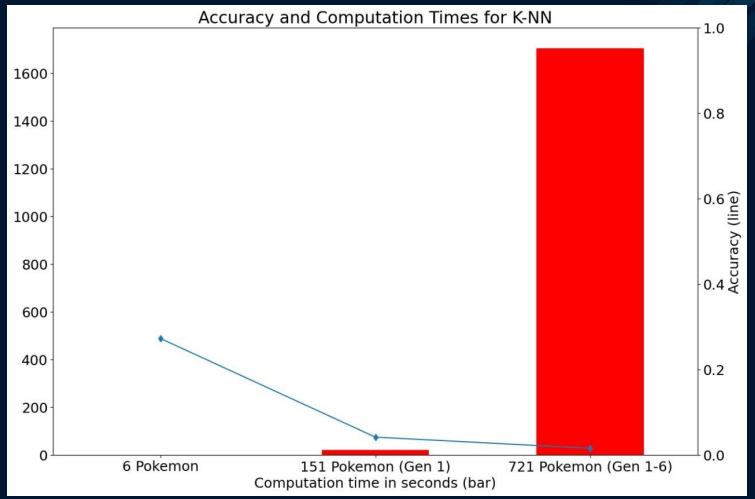
Source:
https://www.pinterest.com/pin/742953269760209829/

Results

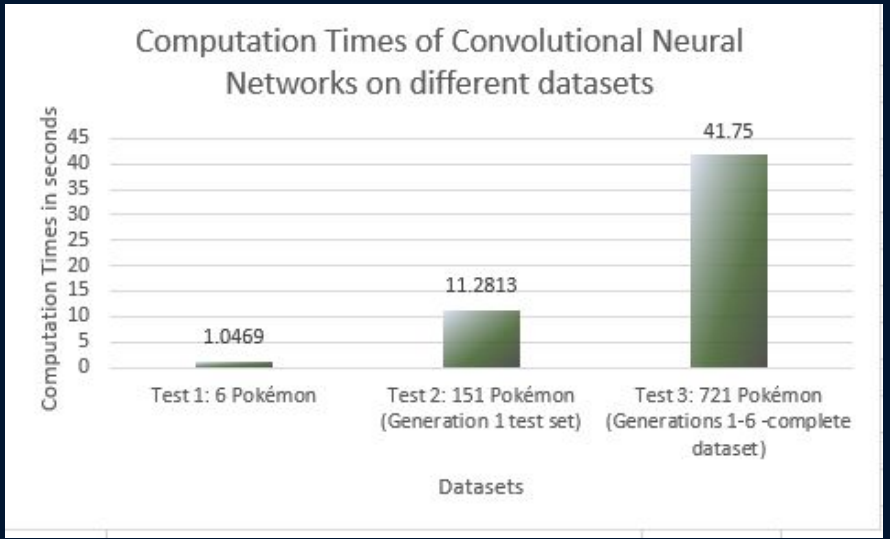

Accuracy and Computation Times for CNN
Accuracy (line)
6 Pokemon
151 Pokemon (Gen 1)
721 Pokemon (Gen 1-6)
Computation time in seconds (bar)

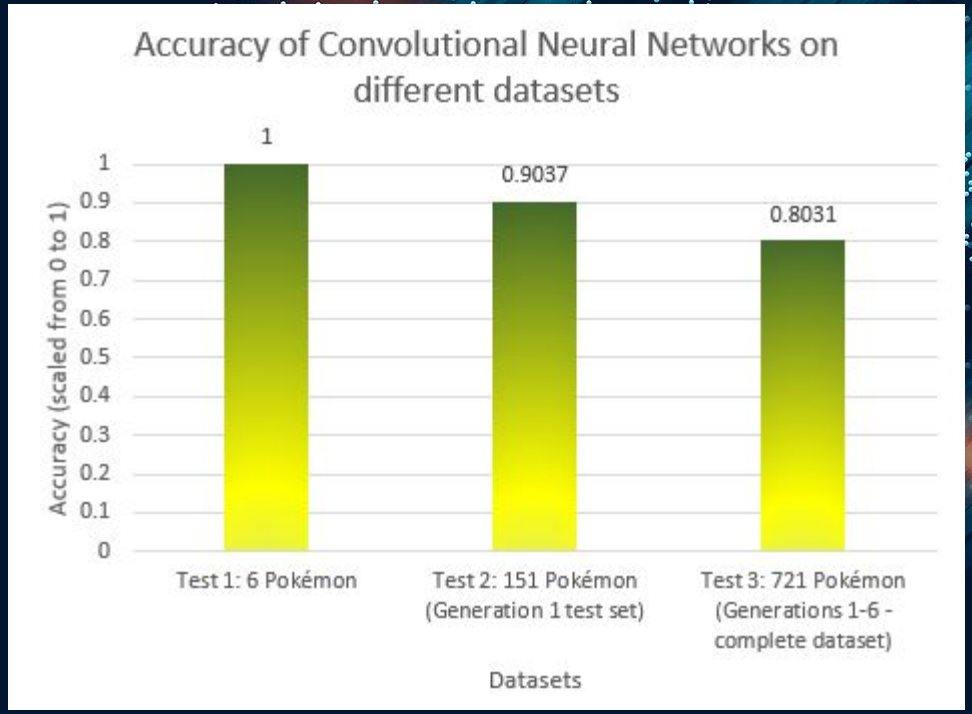

Accuracy and Computation Times for Pre-Trained Models
Accuracy (line)
6 Pokemon
151 Pokemon (Gen 1)
721 Pokemon (Gen 1-6)
Computation time in seconds (bar)


Accuracy and Computation Times for Support Vector Machines
Accuracy (line)
6 Pokemon
151 Pokemon (Gen 1)
721 Pokemon (Gen 1-6)
Computation time in seconds (bar)

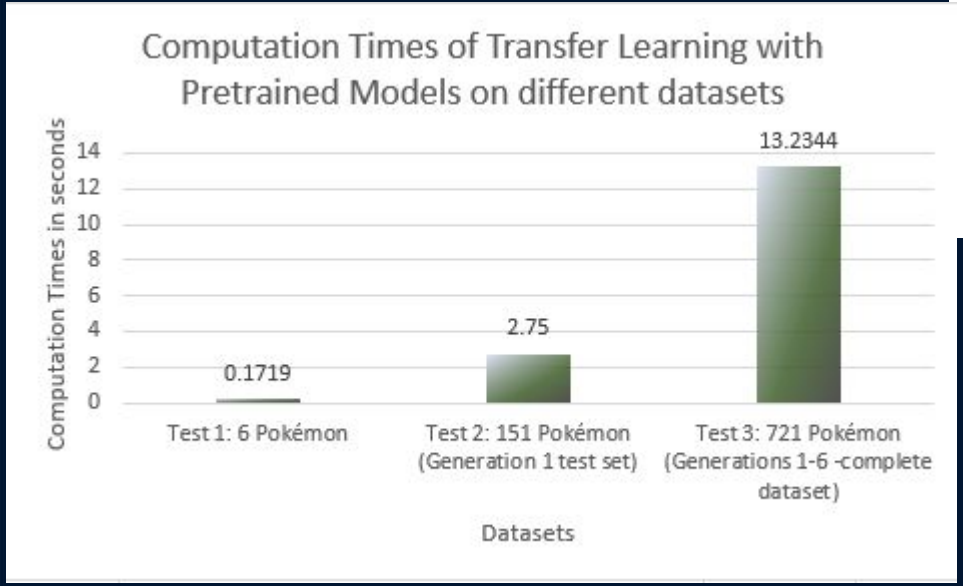

Accuracy and Computation Times for K-NN
Accuracy (line)
6 Pokemon
151 Pokemon (Gen 1)
721 Pokemon (Gen 1-6)
Computation time in seconds (bar)

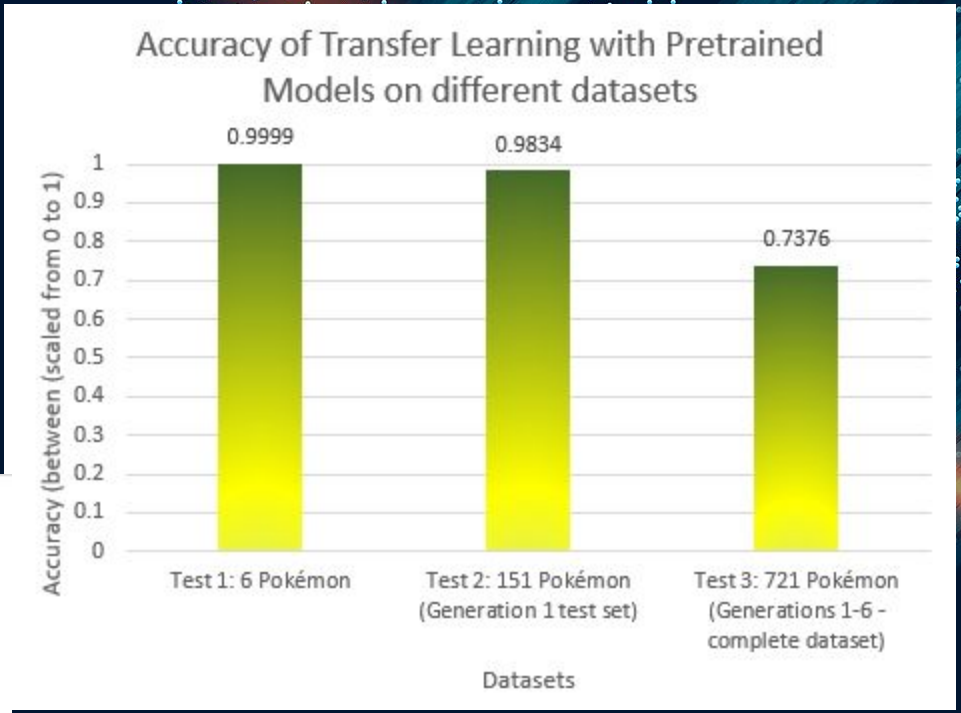
RESULTS FOR CONVOLUTIONAL NEURAL NETWORKS



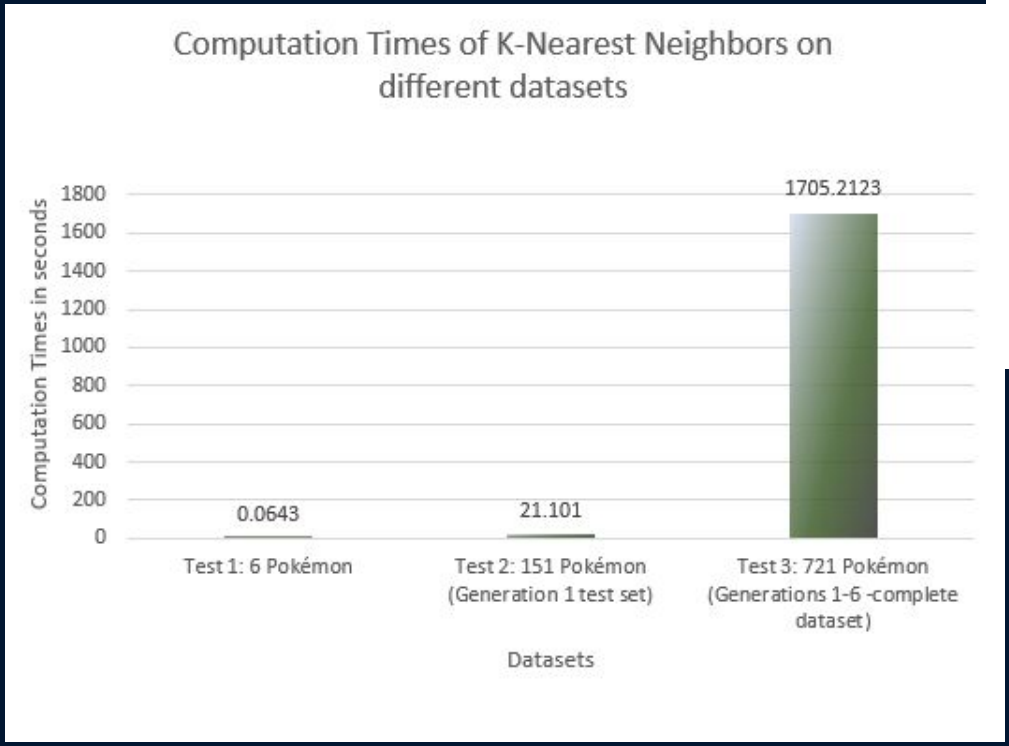


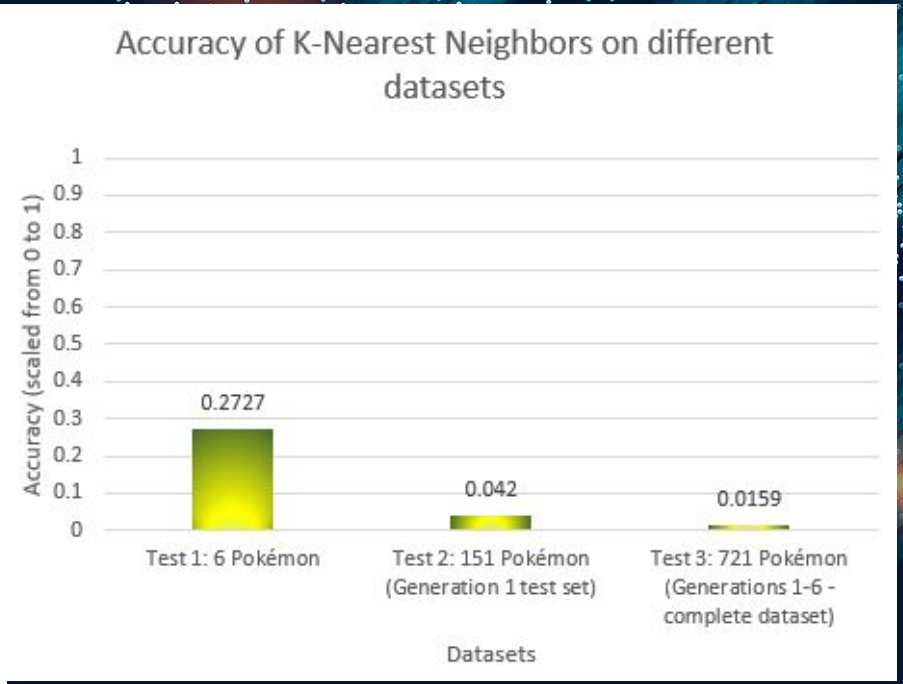
RESULTS FOR TRANSFER LEARNING WITH PRETRAINED MODELS



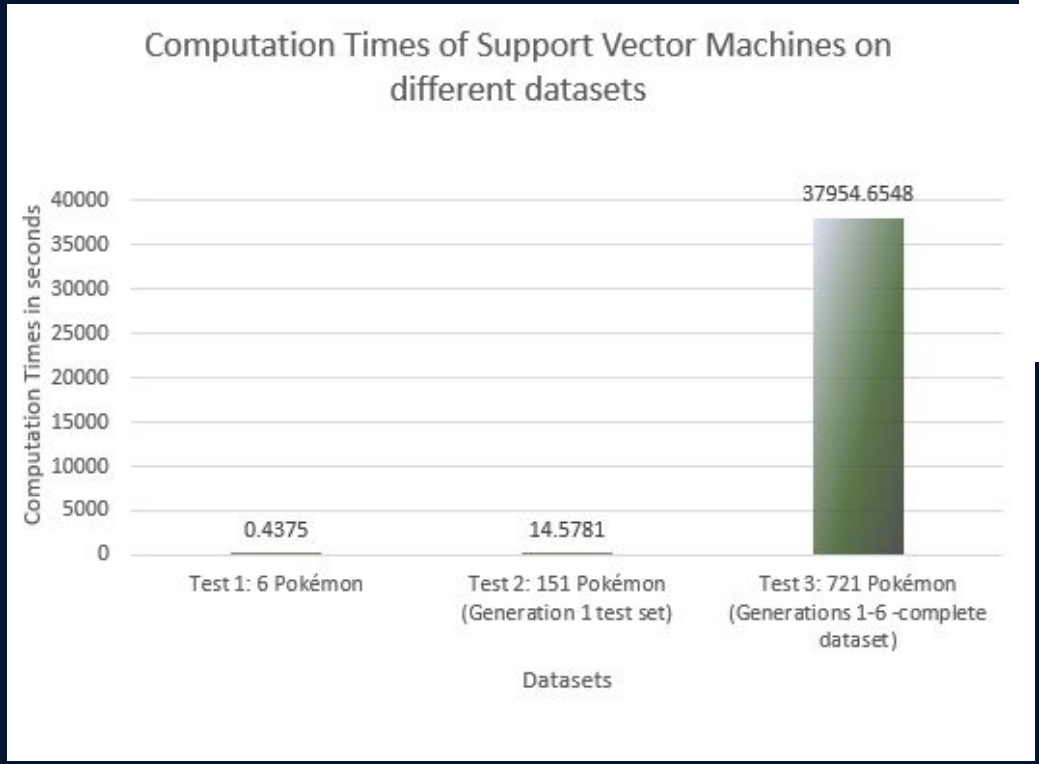


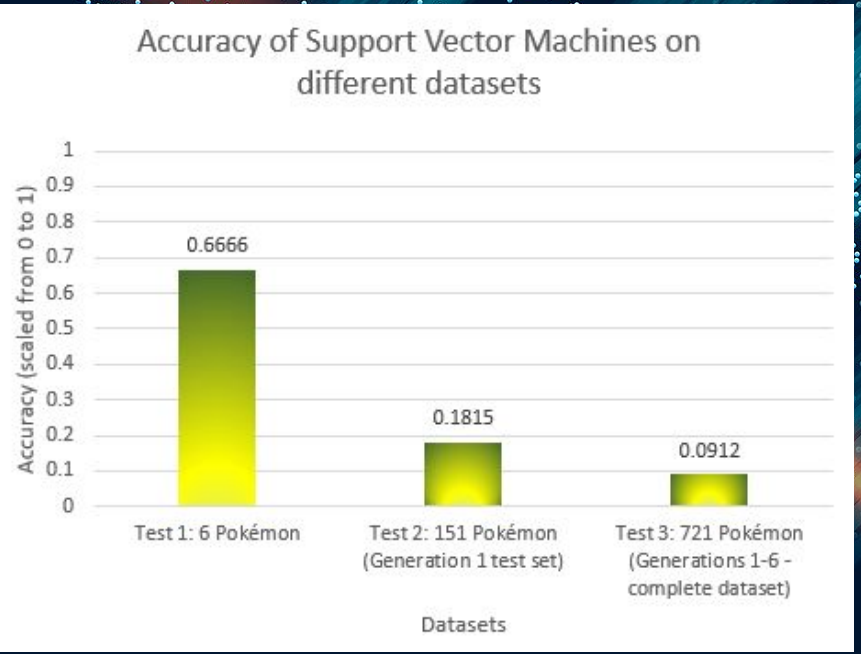
RESULTS FOR K NEAREST NEIGHBORS



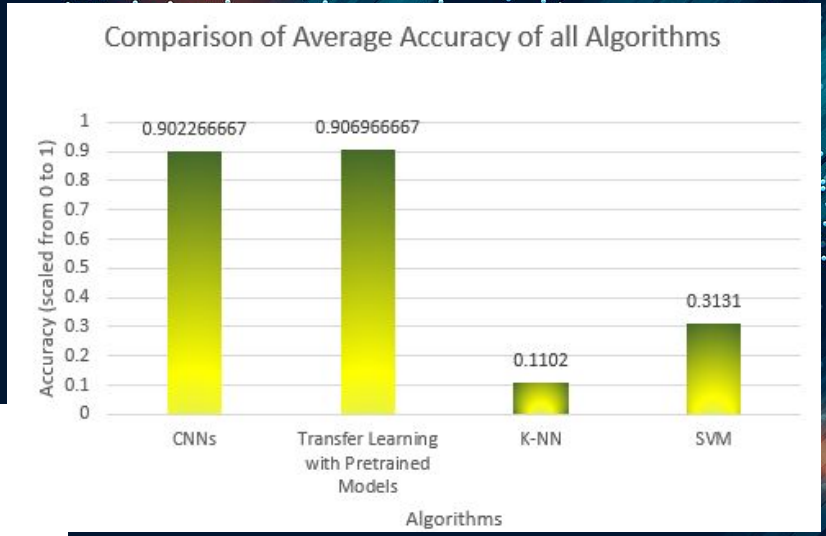


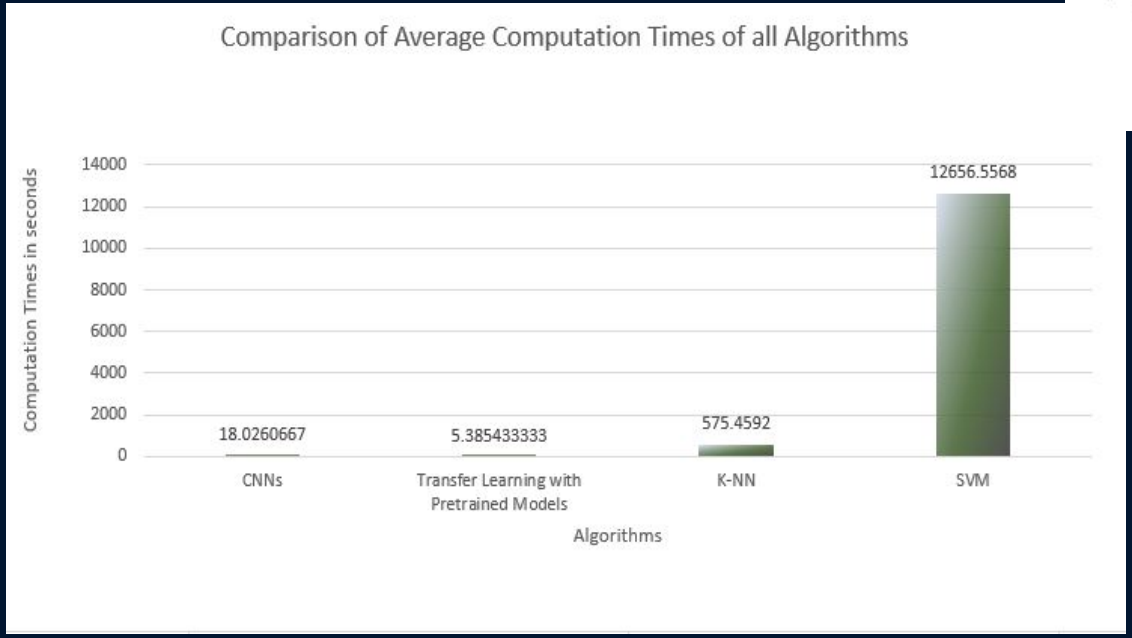
RESULTS FOR SUPPORT VECTOR MACHINES





COMPARISON OF PERFORMANCE OF ALL ALGORITHMS





Conclusions

- Average accuracy and computation time for each algorithm
    - Transfer Learning with Pretrained Models (90.6% and ~5.4 seconds)
    - Convolutional Neural Network (90.2% and ~18.02 seconds)
    - Support Vector Machines (31.31% and ~12656.56 seconds)
    - K-Nearest Neighbor (11.02% and ~575.49 seconds)
- Transfer Learning with Pretrained Models (CNN as Pretrained Model as well as New Classifier) performed the best on both metrics, computation time and accuracy.
- CNN's results were very similar to Transfer Learning on the metric of accuracy, but took longer computation times.
- SVM and K-NN took immensely long to execute as well as provided much poorer performance compared to the Transfer Learning and CNN models because they are not suited for image recognition. They are not good at processing image data, especially large tensors.
- Curse of dimensionality was a problem with K-NN.
- The performance of SVM shows that it is much less suited to large datasets with a large number of classes compared to the other three algorithms.

Questions?