Predicting Winner of a Clash Royale Match

Duy Nhat Vo, Minjoo Kim, Vrushank Agrawal

Our Process

1. Data gathering
2. Data processing
3. Data analysis
4. Basic Algorithms
5. Neural networks
6. Features study

Can we predict the winner of a Clash Royale Match?

MOTIVATION

1 million active daily players.

It is believed that some features in the game can reflect in the outcome of the result and its study is precisely the aim of this project.

Gameplay

- 1v1 real-time (3 mins)
- Deck of 8 cards
- Card costs elixir
- Collect most crowns
- Collect & upgrade cards/towers

The Data

Where do we get it?

We get cards and battles data from APIs listed below through bash scripts like the ones displayed on the right.

https://api.clashroyale.com/v1/players/%23$FILENAME/battlelog - official -> real-time matches

https://royaleapi.github.io/cr-api-data/json/cards.json - unofficial -> detailed card data

~ 54000 battles

```bash

# download battle logs from top players
readarray -t players < ./players.txt
rm -rf ./battles
mkdir ./battles

PLAYER_LIMIT=${#players[@]}

i=1
for player in "${players[@]}"
do
	let tmp=i*100/$PLAYER_LIMIT
	let prog=tmp/5
	let i++
	printf "Downloading: ["
	for ((j=0; j<$prog; j++))
	do
		printf '='
	done
	for ((j=$prog; j < 20; j++))
	do
		printf '-'
	done
	printf "] ($tmp%%) \r"
	FILENAME=$(echo $player | cut -d '#' -f 2)
	curl -H "Authorization: Bearer $API_KEY" "https://api.clashroyale.com/v1/players/%23$FILENAME/battlelog" > "./battles/$FILENAME.json" 2> /dev/null

	# remove players with empty battle logs
	if test $(du -k "./battles/$FILENAME.json" | cut -f 1) -lt 1
	then
		rm -f "./battles/$FILENAME.json"
	fi
done
printf '\n'

# count how many players were downloaded
echo "$(ls ./battles | wc -l) players downloaded"

```

```bash
#!/bin/bash

echo "Downloading cards"

# download cards and stats
curl https://royaleapi.github.io/cr-api-data/json/cards.json > ./cards.json
curl https://royaleapi.github.io/cr-api-data/json/cards_stats.json > ./cards_stats.json
```

The Data

How do we use it?

We process the data in a Jupyter Notebook to update column values, add necessary columns, and filter out incomplete data points.

We convert the data from JSON to CSV for better readability and ease of use.

1	key	name	sc_key	elixir	type	rarity	id	flying_height	range	damage_air	damage_ground
2	knight	Knight	Knight	3	Troop	Common	26000000	0	1200	False	True
3	archers	Archers	Archer	3	Troop	Common	26000001	0	5000	True	True
4	goblins	Goblins	Goblins	2	Troop	Common	26000002	0	500	False	True
5	giant	Giant	Giant	5	Troop	Rare	26000003	0	1200	False	True
6	pekka	P.E.K.K.A	Pekka	7	Troop	Epic	26000004	0	1200	False	True
7	minions	Minions	Minions	3	Troop	Common	26000005	1500	1600	True	True
8	balloon	Balloon	Balloon	5	Troop	Epic	26000006	3000	100	False	True
9	witch	Witch	Witch	5	Troop	Epic	26000007	0	5500	True	True
10	barbarians	Barbarians	Barbarians	5	Troop	Common	26000008	0	700	False	True

	p1_trophy	p1_card_0_id	p1_card_0_lv	p1_card_1_id	p1_card_1_lv	p1_card_2_id	p1_card_2_lv	p1_card_3_id	p1_card_3_lv	p1_card_4_id	...	p2_c
count	54558.000000	5.455800e+04	54558.000000	5.455800e+04	54558.000000	5.455800e+04	54558.000000	5.455800e+04	54558.000000	5.455800e+04	...	5.455
mean	4005.229536	2.640762e+07	6.592415	2.646853e+07	6.686407	2.650099e+07	6.805015	2.666058e+07	6.954617	2.664013e+07	...	2.664
std	1651.431390	7.764509e+05	3.670801	8.155463e+05	3.612161	8.353977e+05	3.690447	9.089260e+05	3.635645	8.966986e+05	...	9.000
min	23.000000	2.600000e+07	1.000000	2.600000e+07	1.000000	2.600000e+07	1.000000	2.600000e+07	1.000000	2.600000e+07	...	2.600
25%	3000.000000	2.600001e+07	4.000000	2.600001e+07	4.000000	2.600001e+07	4.000000	2.600002e+07	4.000000	2.600002e+07	...	2.600
50%	3914.000000	2.600003e+07	6.000000	2.600003e+07	6.000000	2.600003e+07	6.000000	2.600004e+07	7.000000	2.600004e+07	...	2.600
75%	5734.000000	2.600006e+07	9.000000	2.700000e+07	9.000000	2.700001e+07	9.000000	2.800000e+07	9.000000	2.800000e+07	...	2.800
max	7268.000000	2.800002e+07	14.000000	2.800002e+07	14.000000	2.800002e+07	14.000000	2.800002e+07	14.000000	2.800002e+07	...	2.800

8 rows × 37 columns

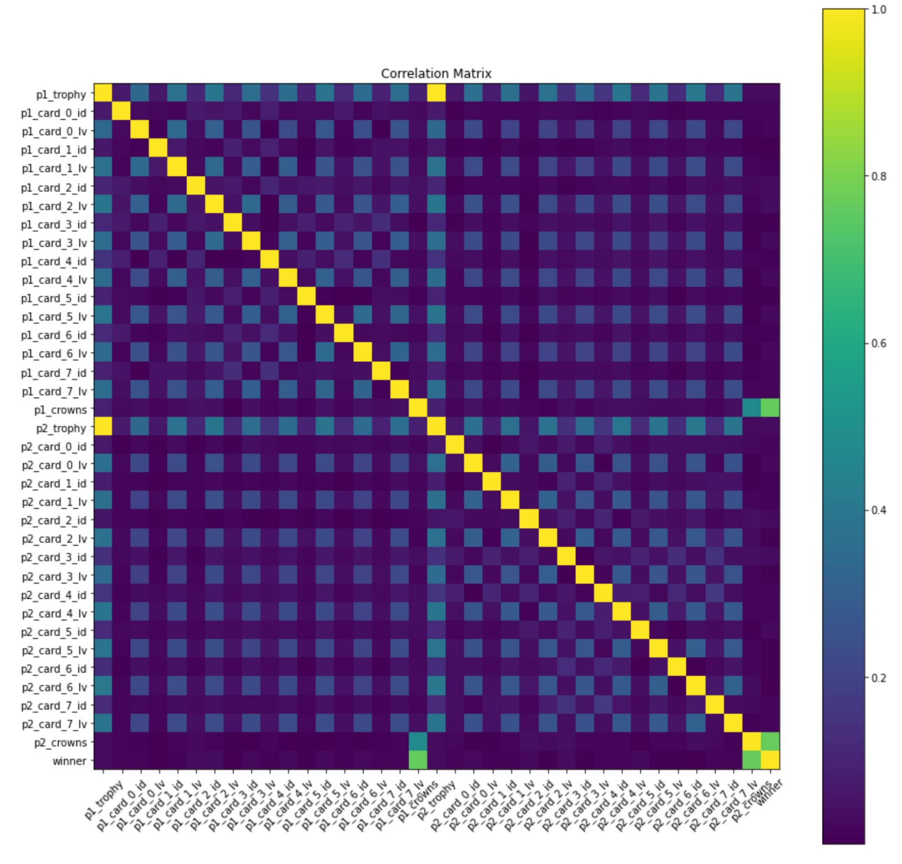
Data Analysis

- Correlations in cards
- Feature analysis
- PCA

Correlation

We observed the correlation between the cards of the players, their levels, and their rankings.

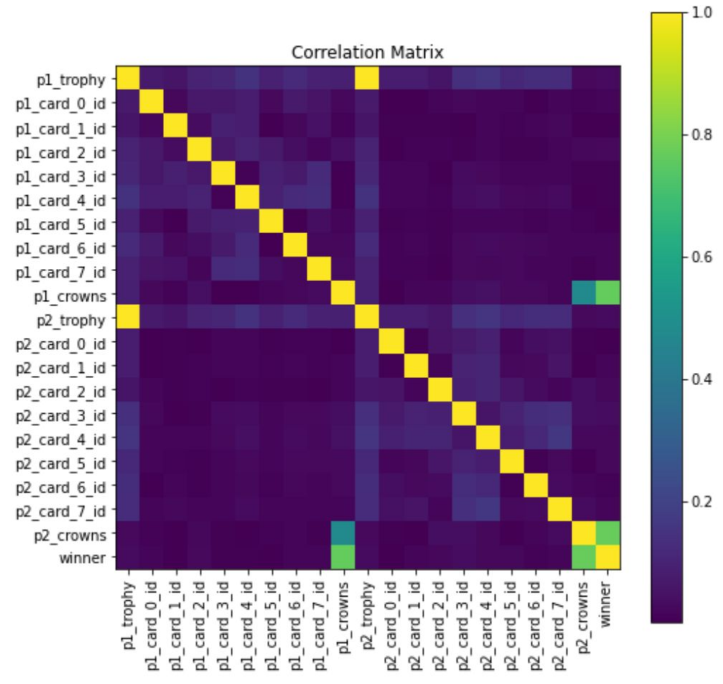
There is clear correlation between cards and their levels, winner and crowns, and trophy and card level.

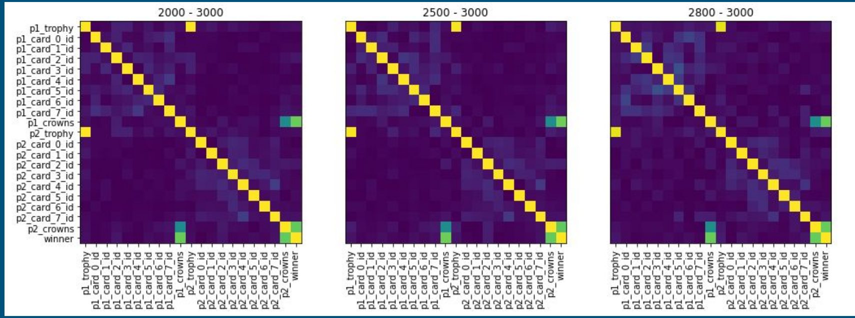


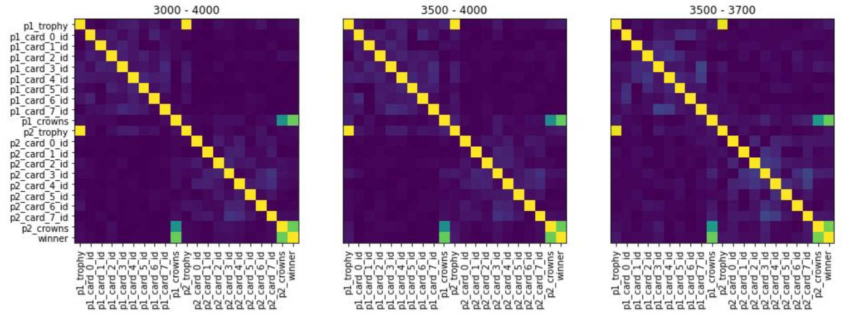
Correlation b/w cards

On a closer look, it is evident that there is some correlation between the card id's in a players deck while there is no correlation between the cards of the two players.

This makes sense and shows that certain cards have synergies and played together while the two players in almost all battles have different decks.







Card Features

Elixir, Type, Rarity

- **Average Deck Elixir**
- **Card Type**
- **Card Rarity**

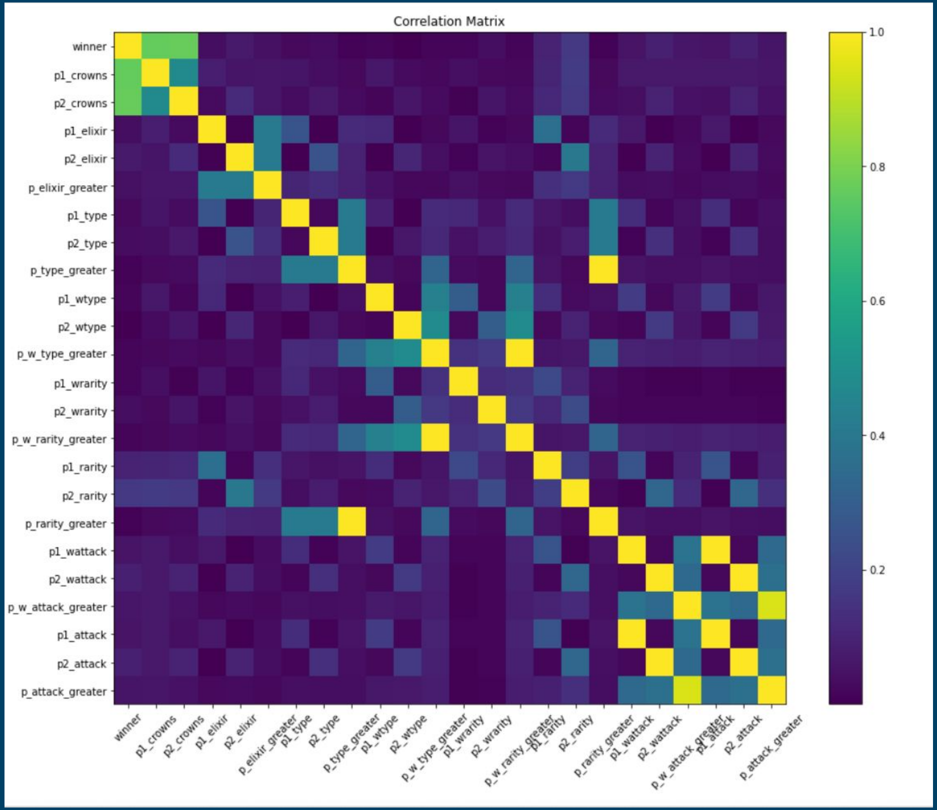
Average Deck Elixir: In Clash Royale, every card has an elixir value which can be thought of as the cost to play that card in the game. This elixir is generated at a certain speed for every player and by playing a specific card, the elixir of that card is lost from the player's total elixir amount. This feature ensures that a player cannot play as many cards as he/she wants. In particular, this feature allows an opportunity for us to study the relation between the average elixir weight of a player's deck to the outcome of the game. In other words, our aim is to study if the player with a cheaper deck has a higher chance of winning the game. If this is the case, then we will have a correlation between the winner and the player who has cheaper deck.

Card Type: Every card in the game also has a type which can be thought of as a class of the card if it is a `Troop`, `Building`, or a `Spell`, and the cards do what the name of their class suggests. In particular, this feature may/may not have a relation between the type of cards of a player's deck and the outcome of the game. In other words, our aim is to study if a certain type of deck has a higher chance of winning the game. If this is the case, then we will have a correlation between the winner and the deck type. For our analysis, we will divide the `Troop` type in two categories `Air Troop` and `Ground Troop`.

Card Rarity: Furthermore, every card in the game also has a rarity which can be thought of as the abundance of the card in the game. Essentially there are 5 categories: `Common`, `Rare`, `Epic`, `Legendary`, and `Champion`. As the names suggest, these cards are decreasingly abundant in the game and it is harder to find these cards to upgrade their levels and play them. In particular, this is another feature that may/may not have a relation between the rarity of cards in a player's deck and the outcome of the game. In other words, our aim is to study if a certain rarity of a deck has a higher chance of winning the game. If this is the case, then we will have a correlation between the winner and the deck rarity.

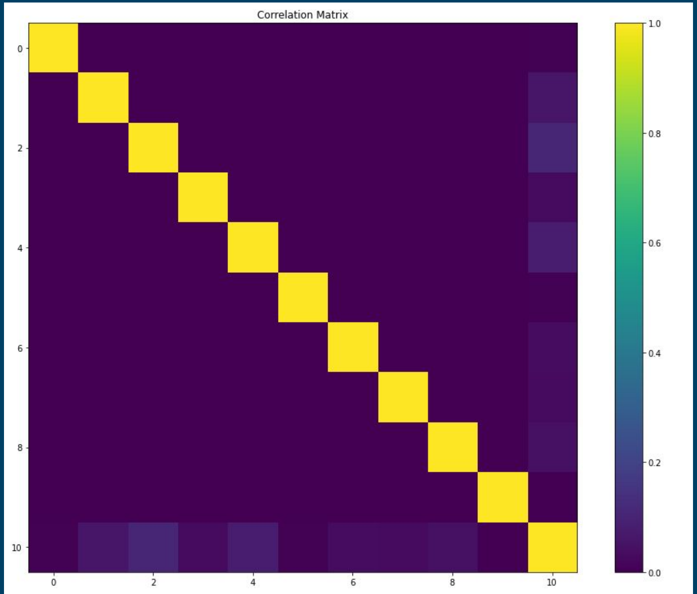
Card Feature Correlations

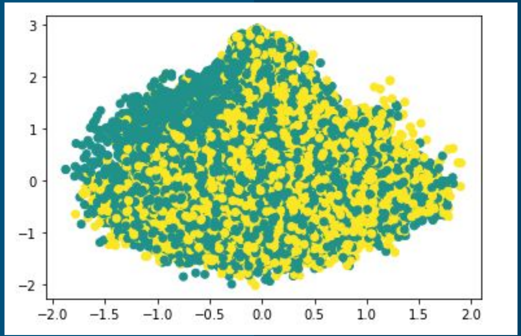
- Card type and elixir
- Card rarity and elixir
- Air troops (attack higher) and elixir/type/rarity

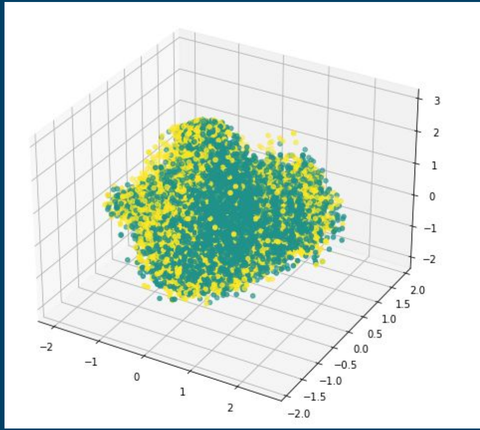


PCA

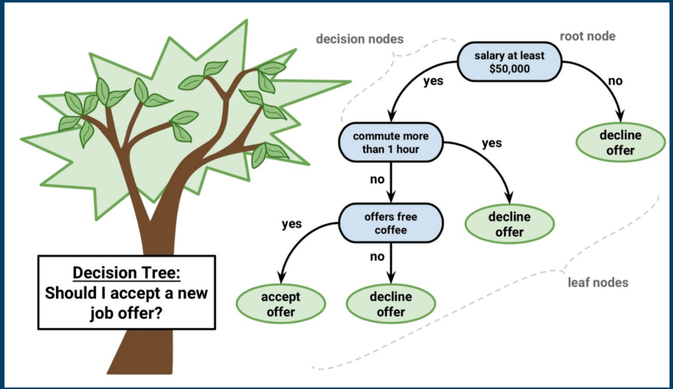
- Select 5/10 components



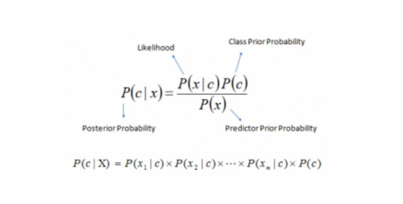


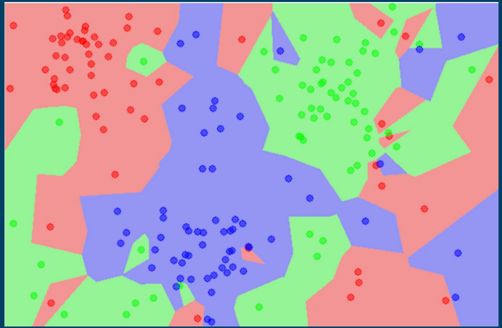


Basic Algorithms



- KNearestNeighbors
- Decision Trees
- Naive Bayes





Results

- Decision Trees and KNN gives around 60%
- PCA does not give a better result

```
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
gnb.fit(x_train, y_train)
print('Training accuracy:\t', gnb.score(x_train, y_train))
print('Testing accuracy:\t', gnb.score(x_test, y_test))
```

```
Training accuracy:	0.5814537137048097
Testing accuracy:	0.5744599745870393
```

```
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(x_train, y_train)
print('Training accuracy:\t', knn.score(x_train, y_train))
print('Testing accuracy:\t', knn.score(x_test, y_test))
```

```
Training accuracy:	0.7260688264674884
Testing accuracy:	0.5667155425219942
```

```
ada = AdaBoostClassifier(DecisionTreeClassifier(max_depth=10))
ada.fit(x_train, y_train)
print('Training accuracy:\t', ada.score(x_train, y_train))
print('Testing accuracy:\t', ada.score(x_test, y_test))
```

```
Training accuracy:	0.774206112816753
Testing accuracy:	0.531524926686217
```

```
bagging = BaggingClassifier(DecisionTreeClassifier(max_depth=10))
bagging.fit(x_train, y_train)
print('Training accuracy:\t', bagging.score(x_train, y_train))
print('Testing accuracy:\t', bagging.score(x_test, y_test))
```

```
Training accuracy:	0.6615955643128809
Testing accuracy:	0.5721224340175953
```

```
extra = ExtraTreesClassifier(max_depth=10)
extra.fit(x_train, y_train)
print('Training accuracy:\t', extra.score(x_train, y_train))
print('Testing accuracy:\t', extra.score(x_test, y_test))
```

```
Training accuracy:	0.6651926866150392
Testing accuracy:	0.5812866568914956
```

```
rf = RandomForestClassifier(max_depth=10)
rf.fit(x_train, y_train)
print('Training accuracy:\t', rf.score(x_train, y_train))
print('Testing accuracy:\t', rf.score(x_test, y_test))
```

```
Training accuracy:	0.6519497777574119
Testing accuracy:	0.5841275659824047
```

With PCA

- Testing score is lower
- Decreasing `n_components` does not work either

```
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(x_train_pca, y_train)
print('Training accuracy:\t', knn.score(x_train_pca, y_train))
print('Testing accuracy:\t', knn.score(pca.transform(x_test), y_test))
```

```
Training accuracy:	0.7166292443752005
Testing accuracy:	0.5439882697947214
```

```
ada = AdaBoostClassifier(DecisionTreeClassifier(max_depth=10))
ada.fit(x_train_pca, y_train)
print('Training accuracy:\t', ada.score(x_train_pca, y_train))
print('Testing accuracy:\t', ada.score(pca.transform(x_test), y_test))
```

```
Training accuracy:	0.8935526737845393
Testing accuracy:	0.5267595307917888
```

```
extra = ExtraTreesClassifier(max_depth=10)
extra.fit(x_train_pca, y_train)
print('Training accuracy:\t', extra.score(x_train_pca, y_train))
print('Testing accuracy:\t', extra.score(pca.transform(x_test), y_test))
```

```
Training accuracy:	0.6160472895568895
Testing accuracy:	0.5601173020527859
```

```
rf = RandomForestClassifier(max_depth=10)
rf.fit(x_train_pca, y_train)
print('Training accuracy:\t', rf.score(x_train_pca, y_train))
print('Testing accuracy:\t', rf.score(pca.transform(x_test), y_test))
```

```
Training accuracy:	0.686202630252486
Testing accuracy:	0.562133431085044
```

```
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
gnb.fit(x_train_pca, y_train)
print('Training accuracy:\t', gnb.score(x_train_pca, y_train))
print('Testing accuracy:\t', gnb.score(pca.transform(x_test), y_test))
```

```
Training accuracy:	0.5516198506163222
Testing accuracy:	0.5559934017595308
```

Additional Features

- We added `type`, `elixir`, and `rarity`
- The result is slightly better

```python
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(x_train, y_train)
print('Training accuracy:\t', knn.score(x_train, y_train))
print('Testing accuracy:\t', knn.score(x_test, y_test))
```

```
Training accuracy:	0.7409582276791672
Testing accuracy:	0.5939167556029883
```

```python
ada = AdaBoostClassifier(DecisionTreeClassifier(max_depth=10))
ada.fit(x_train, y_train)
print('Training accuracy:\t', ada.score(x_train, y_train))
print('Testing accuracy:\t', ada.score(x_test, y_test))
```

```
Training accuracy:	0.9970639263312425
Testing accuracy:	0.5736392742796158
```

```python
bagging = BaggingClassifier(DecisionTreeClassifier(max_depth=10))
bagging.fit(x_train, y_train)
print('Training accuracy:\t', bagging.score(x_train, y_train))
print('Testing accuracy:\t', bagging.score(x_test, y_test))
```

```
Training accuracy:	0.7587081275857467
Testing accuracy:	0.5939167556029883
```

```python
extra = ExtraTreesClassifier(max_depth=10)
extra.fit(x_train, y_train)
print('Training accuracy:\t', extra.score(x_train, y_train))
print('Testing accuracy:\t', extra.score(x_test, y_test))
```

```
Training accuracy:	0.8237021219805152
Testing accuracy:	0.6008537886872999
```

```python
rf = RandomForestClassifier(max_depth=10)
rf.fit(x_train, y_train)
print('Training accuracy:\t', rf.score(x_train, y_train))
print('Testing accuracy:\t', rf.score(x_test, y_test))
```

```
Training accuracy:	0.8066195115441078
Testing accuracy:	0.6104589114194237
```

Neural Networks

- Dense Layers
- Convolutional Layers (1D)

Neural Network

- Dense Network gives a 50% accuracy - 52% with PCA
- Add Convolutional layer -> 55%
- Additional features give very slightly better results
- Add Dense layer to remap input -> not much improvement
- We need to find a better way to represent the features in Neural Network

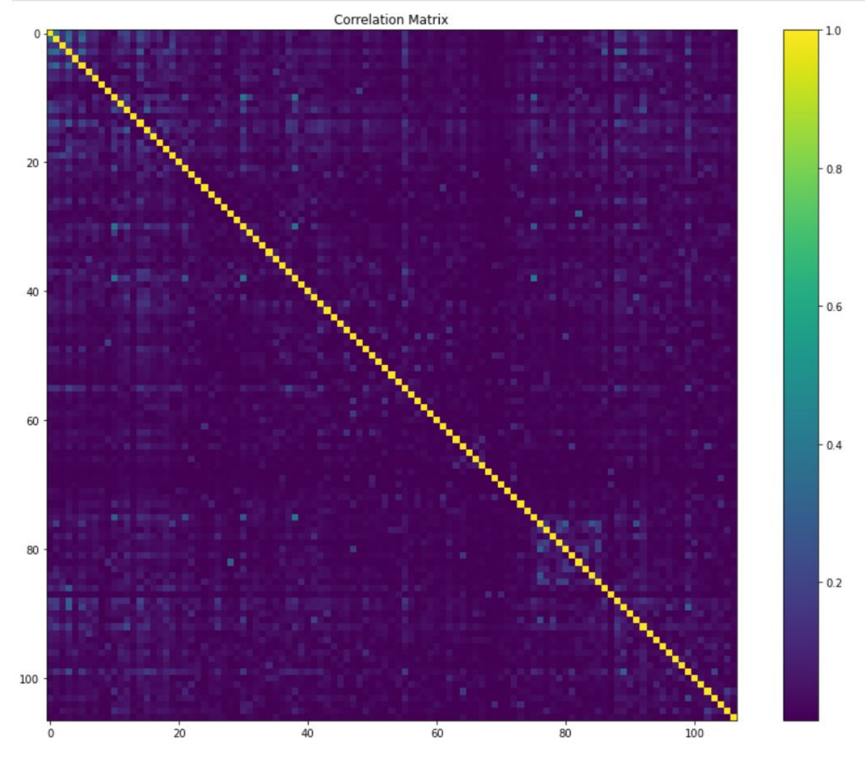
```
cnn.summary()

Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_6 (Dense)              (None, 214, 856)          1712
_________________________________________________________________
conv1d_4 (Conv1D)            (None, 214, 32)           219168
_________________________________________________________________
conv1d_5 (Conv1D)            (None, 207, 32)           8224
_________________________________________________________________
batch_normalization_4 (Batch (None, 207, 32)           128
_________________________________________________________________
max_pooling1d_2 (MaxPooling1 (None, 51, 32)            0
_________________________________________________________________
conv1d_6 (Conv1D)            (None, 44, 64)            16448
_________________________________________________________________
conv1d_7 (Conv1D)            (None, 37, 64)            32832
_________________________________________________________________
batch_normalization_5 (Batch (None, 37, 64)            256
_________________________________________________________________
dense_7 (Dense)              (None, 37, 128)           8320
_________________________________________________________________
dropout_3 (Dropout)          (None, 37, 128)           0
_________________________________________________________________
max_pooling1d_3 (MaxPooling1 (None, 9, 128)            0
_________________________________________________________________
dense_8 (Dense)              (None, 9, 32)             4128
_________________________________________________________________
dropout_4 (Dropout)          (None, 9, 32)             0
_________________________________________________________________
batch_normalization_6 (Batch (None, 9, 32)             128
_________________________________________________________________
dense_9 (Dense)              (None, 9, 16)             528
_________________________________________________________________
dropout_5 (Dropout)          (None, 9, 16)             0
_________________________________________________________________
batch_normalization_7 (Batch (None, 9, 16)             64
_________________________________________________________________
dense_10 (Dense)             (None, 9, 4)              68
_________________________________________________________________
dense_11 (Dense)             (None, 9, 1)              5
=================================================================
Total params: 292,009
Trainable params: 291,721
Non-trainable params: 288
```

Conclusion

- Algorithms give similar results in accuracy
- Features do not seem to contribute strongly to the result
- Possibility on trade-off of features
- Exploiting the different features are difficult (with NN)

Further Study

- Card Synergies
- Deck analysis based on Rankings



Landscape Scene Classification

MOTIVATION

Study the performance of Neural Networks and see if we can recognize different landscapes.

The Data

Where do we get it?

We get the images from the API below through a bash script displayed on the right.

The images need a lot of cleaning and downsampling

https://api.unsplash.com/search/photos?query=$LABEL&page=$i&per_page=1000

```bash
# fetching params
LABEL=$1
PAGES=${2:-10}
START_PAGE=${3:-1}
API_LOC=${4:-'apikey.txt'}

END_PAGE=$(($START_PAGE + $PAGES))
API_KEY=$(head -n 1 $API_LOC)

# params checking
if [ -z "$LABEL" ]
then
        echo "Put in a category (forest, building, sea, mountain, dessert, city)"
    echo "Usage: $0 <categoory> <num. of pages> <start page> <API key file>"
        exit
fi

if [ -z "$API_KEY" ]
then
        echo "Put your API key in apikey.txt"
        exit
fi


# initialize
[ -d "images2/$LABEL" ] && echo "The directory already exists. Moving it to a new one..." && mv "images2/$LABEL" "images2/$LABEL-copied-$START_PAGED"
rm -f photos.txt


# fetch what to download
for ((i = $START_PAGE; i < $END_PAGE; i++))
do
        curl -H "Authorization: Client-ID $API_KEY" \
        "https://api.unsplash.com/search/photos?query=$LABEL&page=$i&per_page=1000" 2> /dev/null > tmp.txt
        cat tmp.txt | tr ',' '\n' | grep thumb | cut -d '"' -f 4 | cut -d '?' -f 1 >> photos.txt
done

```

```bash

# start download
i=1
TOTAL=$(cat photos.txt | wc -l)
echo "Downloading $TOTAL images to images2"

cat photos.txt | while read -r url
do
    # download
    curl -o "images2/$LABEL/$i.jpg" --create-dirs "$url?w=200&h=200&fm=jpg&fit=max" 2> /dev/null

    # progress bar logic
        tmp=$((tmp=i*100/$TOTAL))
        prog=$((prog=tmp/5))
        printf "Downloading: ["
        for ((j=0; j<$prog; j++))
        do
                printf '='
        done
        for ((j=$prog; j < 20; j++))
        do
                printf '-'
        done
        printf "] ($i/$TOTAL) \r"
        i=$((i+1))
done

# clean up
rm -f tmp.txt
echo "Done"
```

The Data

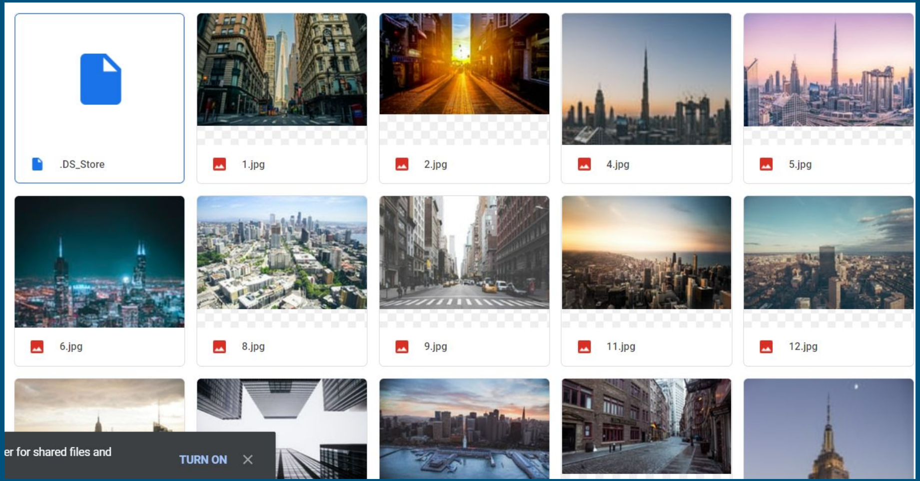
How do we use it?

We process the images into six labels:

'city', 'desert', 'forest',

'ice', 'mountain', 'sea'

Keras' `ImageDataGenerator` is very useful for generating image samples




```
imggen=image.ImageDataGenerator(
    rescale=1/255,
    rotation_range=20,
    zoom_range=0.2,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True,
    validation_split=0.2
)
```

KNN and Decision Trees

Using these algorithms alone is hard to get a good results

```
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(x_train, y_train)
print('Training accuracy:\t', knn.score(x_train, y_train))
print('Testing accuracy:\t', knn.score(x_test, y_test))
```

```
Training accuracy:      0.493
Testing accuracy:       0.2833333333333333
```

```
ada = AdaBoostClassifier(DecisionTreeClassifier(max_depth=10))
ada.fit(x_train, y_train)
print('Training accuracy:\t', ada.score(x_train, y_train))
print('Testing accuracy:\t', ada.score(x_test, y_test))
```

```
Training accuracy:      1.0
Testing accuracy:       0.395
```

```
bagging = BaggingClassifier(DecisionTreeClassifier(max_depth=10))
bagging.fit(x_train, y_train)
print('Training accuracy:\t', bagging.score(x_train, y_train))
print('Testing accuracy:\t', bagging.score(x_test, y_test))
```

```
Training accuracy:      0.9046666666666666
Testing accuracy:       0.3433333333333333
```

```
extra = ExtraTreesClassifier(max_depth=10)
extra.fit(x_train, y_train)
print('Training accuracy:\t', extra.score(x_train, y_train))
print('Testing accuracy:\t', extra.score(x_test, y_test))
```

```
Training accuracy:      0.9183333333333333
Testing accuracy:       0.4166666666666667
```

```
rf = RandomForestClassifier(max_depth=10)
rf.fit(x_train, y_train)
print('Training accuracy:\t', rf.score(x_train, y_train))
print('Testing accuracy:\t', rf.score(x_test, y_test))
```

```
Training accuracy:      0.9326666666666666
Testing accuracy:       0.4166666666666667
```

```
gnb = GaussianNB()
gnb.fit(x_train, y_train)
print('Training accuracy:\t', gnb.score(x_train, y_train))
print('Testing accuracy:\t', gnb.score(x_test, y_test))
```

```
Training accuracy:      0.37
Testing accuracy:       0.35333333333333333
```

Neural Network

We compare several neural networks and get the best results for `AlexNet` with ~79% accuracy.

LeNet

```
Epoch 7/10
1136/1136 [==============================] - 35s 31ms/step - loss: 1.1068 - accuracy: 0.5745 - val_loss: 1.0875 - val_accuracy: 0.5944
Epoch 8/10
1136/1136 [==============================] - 34s 30ms/step - loss: 1.0855 - accuracy: 0.5791 - val_loss: 1.0740 - val_accuracy: 0.5944
Epoch 9/10
1136/1136 [==============================] - 35s 31ms/step - loss: 1.0684 - accuracy: 0.5918 - val_loss: 1.0741 - val_accuracy: 0.6032
Epoch 10/10
1136/1136 [==============================] - 36s 32ms/step - loss: 1.0587 - accuracy: 0.5935 - val_loss: 1.0966 - val_accuracy: 0.5911
<keras.callbacks.History at 0x7f0c88dcf3d0>
```

Custom

```
Epoch 7/10
1136/1136 [==============================] - 249s 219ms/step - loss: 0.6741 - accuracy: 0.7532 - val_loss: 1.8683 - val_accuracy: 0.4928
Epoch 8/10
1136/1136 [==============================] - 247s 218ms/step - loss: 0.6411 - accuracy: 0.7654 - val_loss: 0.9595 - val_accuracy: 0.6296
Epoch 9/10
1136/1136 [==============================] - 247s 218ms/step - loss: 0.6257 - accuracy: 0.7732 - val_loss: 1.0580 - val_accuracy: 0.6311
Epoch 10/10
1136/1136 [==============================] - 250s 220ms/step - loss: 0.5965 - accuracy: 0.7901 - val_loss: 0.8919 - val_accuracy: 0.6921
<keras.callbacks.History at 0x7f118d536d50>
```

AlexNet

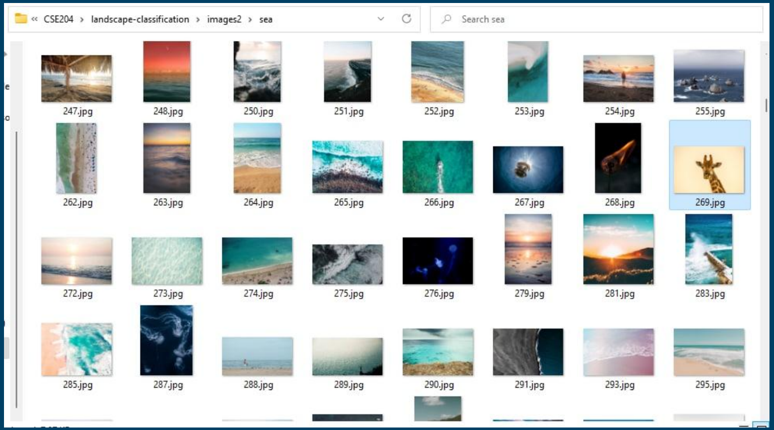
```
Epoch 7/10
1136/1136 [==============================] - 207s 182ms/step - loss: 0.7437 - accuracy: 0.7228 - val_loss: 0.6494 - val_accuracy: 0.7646
Epoch 8/10
1136/1136 [==============================] - 203s 179ms/step - loss: 0.6992 - accuracy: 0.7432 - val_loss: 0.7896 - val_accuracy: 0.7156
Epoch 9/10
1136/1136 [==============================] - 203s 179ms/step - loss: 0.6695 - accuracy: 0.7543 - val_loss: 0.6587 - val_accuracy: 0.7652
Epoch 10/10
1136/1136 [==============================] - 200s 176ms/step - loss: 0.6342 - accuracy: 0.7677 - val_loss: 0.5979 - val_accuracy: 0.7875
<keras.callbacks.History at 0x7f0c88e46410>
```

Conclusion

We have been able to product acceptable results with the data which can classify the six labels with 79% accuracy.

Further Study

- Filtering misclassified data
- Analyze confusing categories
- Create precise algorithms for this specific case of classification.




```
from sklearn.metrics import confusion_matrix
print(confusion_matrix(y_test, y_pred))
```

```
[[130  15  33   9   7  16]
 [  3 174   6  22   1  14]
 [  3   5 177   3  16   7]
 [  1   3   0 174  11   4]
 [  1   3   9  12 188   5]
 [  4  38  11   9  15  71]]
```

```
val_data.class_indices
```

```
{'beach': 0,
 'city': 1,
 'desert': 2,
 'forest': 3,
 'grassland': 4,
 'mountains': 5}
```

Code

Clash Royale:
https://github.com/nhat-vo/clash-royale-ml

Scene Classification:
https://github.com/nhat-vo/landscape-classification

Classification Dataset:
https://drive.google.com/drive/folders/18fF716H6f32H14fiULOZZ1EKWdYHbh85?usp=sharing