Interpreting Genome using ML model for Comparison of Genomes.

Nadilla Yaswanth Baba

INTRODUCTION

A genome is a complete collection of DNA in an organism. All living species possess a genome, but they differ considerably in size. The human genome is arranged into 23 chromosomes.

A human genome has about 6 billion characters or letters. If Genome is considered as a book, then it is like a book with about 6 billion letters of “A”, “C”, “G” and “T”. Everyone has a unique genome. Nevertheless, scientists find that most parts of the human genomes are identical to each other.

Genomics comprehensively uses Machine Learning to capture needs in data and deduce new biological hypotheses. By effectively leveraging large data sets, deep learning has reconstructed fields such as computer vision and natural language processing. It has become the method of first choice for many genomics modelling tasks, including predicting the impact of genetic variation on gene regulatory mechanisms like DNA receptiveness and splicing.

So, in this project we are going to explore different methods in python that are helpful to interpret genome in different ways and, we will understand how machine learning algorithms can be used to build a prediction model on DNA sequence data.

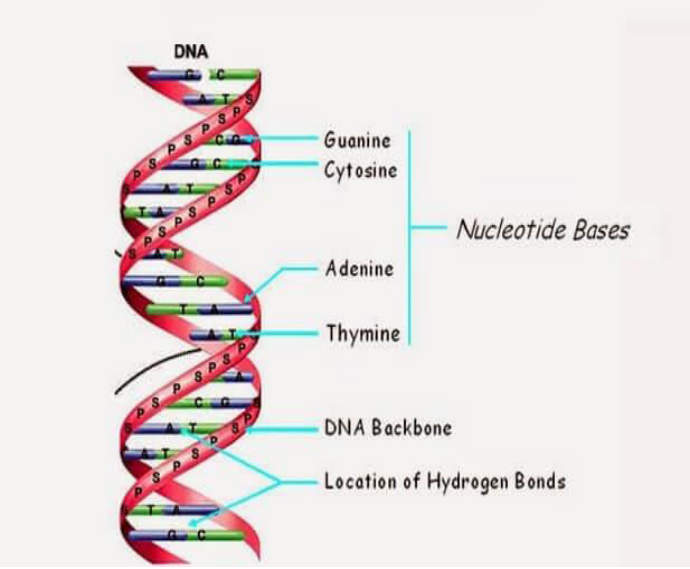
We will also try to predict the closeness of other creature's genomes with Human genome sequence.

There are 3 general approaches to encode sequence data:

1. Ordinal encoding DNA Sequence
2. One-hot encoding DNA Sequence
3. DNA sequence as a "language", known as K-Mer counting

We will be using one of the above-mentioned methods to build the proposed Machine Learning Model.

How is a DNA Sequence represented?



The double-helix is the exact chemical representation of our DNA. But DNA is unique. It's a nucleotide which is made up of four types of nitrogen bases:

Adenine (A), Thymine (T), Guanine (G), and Cytosine. We always call them A, C, G and T.

These four chemicals link together by hydrogen bonds in the possible order making a chain and gives one strand of the DNA double-helix and the second strand of the double-helix balances the existing strand. So, if we have A on the first strand, you have to have T on the second strand.

DNA data handling using Biopython

DNA sequence in fasta format using Biopython. The sequence object will contain attributes such as id and sequence and the length of the sequence that you can work with directly.

We will use Bio.SeqIO from Biopython for parsing DNA sequence data(fasta). It provides a simple uniform interface to input and output assorted sequence file formats.

Sequence Input

The main function is `Bio.SeqIO.parse()` which takes a file handle (or filename) and format name and returns a `SeqRecord` iterator.

Sequence Output

For writing records to a file use the function `Bio.SeqIO.write()`, which takes a SeqRecord iterator (or list), output handle (or filename) and format string.

Now since machine learning models require an input to be in the form of numerical values and currently, we still have our data in string format. So, the later step is to encode these strings into matrices.

The two most widely known techniques are an **Ordinal Encoding** and **One-Hot Encoding**.

Ordinal Encoding

In ordinal encoding, each unique category value is assigned an integer value. It is easily reversible. Often, integer values starting at zero are used.

For some variables, Ordinal encoding may be enough. The integer values have a real arranged relationship between each other, and machine learning algorithms may be able to understand and utilize this relationship.

It is a biological encoding for ordinal variables. For categorical variables, it requires an ordinal relationship where no such relationship exists. This might cause problems and then one-hot encoding may be used instead.

By using default, it's going to assign integers to labels inside the order that is observed within the information. If a selected order is favoured, it may be designated through the “categories” argument as a list with the rank order of all predicted labels.

One-Hot Encoding

For specific variables wherein no ordinal relationship exists, the integer encoding may not be enough.

Forcing an ordinal dating via an ordinal encoding and permitting the version to anticipate a natural ordering among classes may also bring about negative performance or unexpected outcomes.

In this case, a one-hot encoding can be applied to the ordinal illustration. this is wherein the integer encoded variable is eliminated and one new binary variable is added for each unique integer fee in the variable.

If you know all the labels to be expected in the data, they can be specified as the "*categories*" argument as a list.

DNA Sequence as K-mer

We first take the long biological sequence and break it down into k-mer length overlapping "words".

For example, if we use "words" of length 3 (three-mers), "ATGCAT" becomes: 'ATG', 'TGC', 'GCA', 'CAT'. Hence our example sequence is broken down into 4 three-mer words.

In genomics, we refer to these types of manipulations as "k-mer counting" or counting the occurrences of each possible k-mer sequence and Python natural language processing tools make it super easy.

It reverts a list of k-mer "words." We then join the "words" into a "sentence", then use our desired NLP methods on the "sentences" as you normally would.

You can change both the length of the word and the amount of overlap. This allows us to verify how the DNA sequence information and the size will be crucial in your application.

We can then go on and create a bag-of-words model like you would in NLP.

A bag-of-words is a description of text that explains the occurrence of words within a document.

MACHINE LEARNING MODEL

Now that we have learned how to extract feature matrix from the DNA sequence, we will apply our knowledge to a real-life Machine Learning use case.

Objective: Build a classification model that is trained on the human DNA sequence and can predict a gene family based on the DNA sequence of the coding sequence. To test the model, we will use the DNA sequence of humans, dogs, and chimpanzees and compare the accuracies.

Gene families are groups of related genes that share a common ancestor. Members of gene families may be paralogs or orthologs. Gene paralogs are genes with similar sequences from within the same species while gene orthologs are genes with similar sequences in different species.

Then we Load Human DNA data, Chimpanzee DNA data and Dog DNA data.

We define for each of the 7 classes and how many there are in the human training data:

Gene family	Number	Class label
G protein coupled receptors	531	0
Tyrosine kinase	534	1
Tyrosine phosphatase	349	2
Synthetase	672	3
Synthase	711	4
Ion channel	240	5
Transcription factor	1343	6

Now after having all our loaded data, the next step is to convert a sequence of characters into k-mer words with default **size = 6 (hexamers)**.

The **Kmers_funct()** function will collect all the overlapping k-mers of a defined length from any sequence.

The DNA sequence is then changed into lowercase and is divided into all possible k-mer words of length 6 and then converting the lists of k-mers for each gene into string sentences of words that can be used to create the BoW model for all three creatures.

We will make a target variable **y** to hold on all the class labels.

Now that the target variables contain the array of Class values. We use the BoW model using CountVectorizer() which is equivalent to k-mer counting and the size of n-gram will be 4.

Now we will convert our k-mer words into uniform length numerical vectors that shows the count of every k-mer present.

In our project, humans have **4380** genes converted into uniform length feature vectors of 4-gram k-mer (length = 6) counts. Chimpanzee and dog have the same number of features **1682** and **820** genes respectively.

Now we know how to transform our DNA sequences into uniform length vectors in the form of k-mer counts and n-grams, we will now build a classification model that can predict the DNA sequence function established only on sequence itself.

We will use the human data to train the model, keeping 20% of the human data to test the model. Then we can test the model's generalizability by attempting to predict sequence function in chimpanzee and dog.

We use the train/test split human dataset and build simple **Multinomial Naive Bayes classifier**.

MULTINOMIAL NAIVE BAYES

Multinomial Naive Bayes is one of the most popular supervised learning classifications that is used for the analysis of the categorical text data.

It is a probabilistic learning method that is mostly used in Natural Language Processing (NLP). The algorithm is based on the Bayes theorem and predicts the tag of a text such as a piece of email or newspaper article. It calculates the probability of each tag for a given sample and then gives the tag with the highest probability as output.

Naive Bayes classifier is a collection of many algorithms where all the algorithms share one common principle, and that is each feature being

classified is not related to any other feature. The presence or absence of a feature does not affect the presence or absence of the other feature.

It is important to understand the Bayes theorem concept first as it is based on the latter.

Bayes theorem calculates the probability of an event occurring based on the prior knowledge of conditions related to an event. It is based on the following formula:

- $P(A|B) = P(A) * P(B|A)/P(B)$
  - Where we are calculating the probability of class A when predictor B is already provided.
  - P(B) = prior probability of B.
  - P(A) = prior probability of class A
  - P(B|A) = occurrence of predictor B given class A probability

APPLICATIONS OF MULTINOMIAL NAIVE BAYES

- Naive Bayes classifier is used in Text Classification, Spam filtering and Sentiment Analysis. It has a higher success rate than other algorithms.
- Naïve Bayes along with Collaborative filtering are used in Recommended Systems.
- It is also used in disease prediction based on health parameters.
- This algorithm has also found its application in Face recognition.
- Naive Bayes is used in prediction of weather reports based on atmospheric conditions (temp, wind, clouds, humidity etc.)

ADVANTAGES and DISADVANTAGES OF MULTINOMIAL NAIVE BAYES

- It is simple to implement because all you must do is calculate probability. This approach works with both continuous and discrete data. It's straightforward and can be used to forecast real-time applications. It's very scalable and can handle enormous datasets with ease.
- This algorithm's prediction accuracy is lower than that of other probability algorithms. The Naive Bayes technique can only be used to classify textual input and cannot be used to estimate numerical values.

Making predictions on Human hold out test set and performance of unseen data.

Looking at the model performance metrics such as the **confusion matrix, accuracy, precision, recall and f1 score.** We observe we are obtaining really good results on our unseen data, so it looks like our model did a good job and did not overfit to the training data.

Now for the actual test. Let's see how our model performs on the DNA sequences of other species. Firstly, we'll try the Chimpanzee, which we expect to be very similar to that of human. Then we will try it with Humans' best friend, the Dog DNA sequences.

The model produces good results on Human DNA data. It also does on Chimpanzee because the Chimpanzees and humans share identical genetic hierarchy.

The performance of the dog is not as good as the latter which is because the dog is more differing from humans than the chimpanzee.

CODE:

DNA Data handling using Biopython

In [1]:
```python
from Bio import SeqIO
for sequence in SeqIO.parse('C:\\Users\\ashri\\Downloads\\IBS3EndSem Project\\example_
    print(sequence.id)
    print(sequence.seq)
    print(len(sequence))
```

```
ENST00000435737.5
ATGTTTCGCATCACCAACATTGAGTTTCTTCCCGAATACCGACAAAAGGAGTCCAGGGAATTTCTTTCAGTGTCACGGACTGTGCA
GCAAGTGATAAACCTGGTTTATACAACATCTGCCTTCTCCAAATTTTATGAGCAGTCTGTTGTTGCAGATGTCAGCAACAACAAAG
GCGGCCTCCTTGTCCACTTTTGGATTGTTTTTGTCATGCCACGTGCCAAAGGCCACATCTTCTGTGAAGACTGTGTTGCCGCCATC
TTGAAGGACTCCATCCAGACAAGCATCATAAACCGGACCTCTGTGGGGAGCTTGCAGGGACTGGCTGTGGACATGGACTCTGTGGT
ACTAAATGAAGTCCTGGGGCTGACTCTCATTGTCTGGATTGACTGA
390
ENST00000419127.5
ATGTTTCGCATCACCAACATTGAGTTTCTTCCCGAATACCGACAAAAGGAGTCCAGGGAATTTCTTTCAGTGTCACGGACTGTGCA
GCAAGTGATAAACCTGGTTTATACAACATCTGCCTTCTCCAAATTTTATGAGCAGTCTGTTGTTGCAGATGTCAGCAACAACAAAG
GCGGCCTCCTTGTCCACTTTTGGATTGTTTTTGTCATGCCACGTGCCAAAGGCCACATCTTCTGTGAAGACTGTGTTGCCGCCATC
TTGAAGGACTCCATCCAGACAAGCATCATAAACCGGACCTCTGTGGGGAGCTTGCAGGGACTGGCTGTGGACATGGACTCTGTGGT
ACTAAATGACAAAGGCTGCTCTCAGTACTTCTATGCAGAGCATCTGTCTCTCCACTACCCGCTGGAGATTTCTGCAGCCTCAGGGA
GGCTGATGTGTCACTTCAAGCTGGTGGCCATAGTGGGCTACCTGATTCGTCTCTCAATCAAGTCCATCCAAATCGAAGCCGACAAC
TGTGTCACTGACTCCCTGACCATTTACGACTCCCTTTTGCCCATCCGGAGCAGCATCTTGTACAGAATTTGTGAACCCCACAAGAAC
ATTAATGTCATTTGTTTCTACAAATAATCTCATGTTGGTGACATTTAAGTCTCCTCATATACGGAGGCTCTCAGGAATCCGGGCAT
ATTTTGAGGTCATTCCAGAACAAAAGTGTGAAAACACAGTGTTGGTCAAAGACATCACTGGCTTTGAAGGGAAAATTTCAAGCCCA
TATTACCCGAGCTACTATCCTCCAAAATGCAAGTGTACCTGGAAATTTCAGACTTCTCTATCAACTCTTGGCATAGCACTGAAATT
CTATAACTATTCAATAACCAAGAAGAGTATGAAAGGCTGTGAGCATGGATGGTGGGAAATTAATGAGCACATGTACTGTGGCTCCT
ACATGGATCATCAGACAATTTTTCGAGTGCCCAGCCCTCTGGTTCACATTCAGCTCCAGTGCAGTTCAAGGCTTTCAGACAAGCCA
CTTTTGGCAGAATATGGCAGTTACAACATCAGTCAACCCTGCCCTGTTGGATCTTTTAGATGCTCCTCCGGTTTATGTGTCCCTCA
GGCCCAGCGTTGTGATGGAGTAAATGACTGCTTTGATGAAAGTGATGAACTGTTTTGCGTGAGCCCTCAACCTGCCTGCAATACCA
GCTCCTTCAGGCAGCATGGCCCTCTCATCTGTGATGGCTTCAGGGACTGTGAGAATGGCCGGGATGAGCAAAACTGCACTCAAAGT
ATTCCATGCAACAACAGAACTTTTAAGTGTGGCAATGATATTTGCTTTAGGAAACAAAATGCAAAATGTGATGGGACAGTGGATTG
TCCAGATGGAAGTGATGAAGAAGGCTGCACCTGCAGCAGGAGTTCCTCCGCCCTTCACCGCATCATCGGAGGCACAGACACCCTGG
AGGGGGGTTGGCCGTGGCAGGTCAGCCTCCACTTTGTTGGATCTGCCTACTGTGGTGCCTCAGTCATCTCCAGGGAGTGGCTTCTT
TCTGCAGCCCACTGTTTTCATGGAAACAGGCTGTCAGATCCCACACCATGGACTGCACACCTCGGGATGTATGTTCAGGGGAATGC
CAAGTTTGTCTCCCCGGTGAGAAGAATTGTGGTCCACGAGTACTATAACAGTCAGACTTTTGATTATGATATTGCTTTGCTACAGC
TCAGTATTGCCTGGCCTGAGACCCTGAAACAGCTCATTCAGCCAATATGCATTCCTCCCACTGGTCAGAGAGTTCGCAGTGGGGAG
AAGTGCTGGGTAACTGGCTGGGGGCGAAGACACGAAGCAGATAATAAAGGCTCCCTCGTTCTGCAGCAAGCGGAGGTAGAGCTCAT
TGATCAAACGCTCTGTGTTTCCACCTACGGGATCATCACTTCTCGGATGCTCTGTGCAGGCATAATGTCAGGCAAGAGAGATGCCT
GCAAAGGAGATTCGGGTGGACCTTTATCTTGTCGAAGAAAAAGTGATGGAAAATGGATTTTGACTGGCATTGTTAGCTGGGGACAT
GGAAGTGGACGACCAAACTTTCCTGGTGTTTACACAAGGGTGTCAAACTTTGTTCCCTGGATTCATAAATATGTCCCTTCTCTTTT
GTAA
2154
```

So it produces the sequence ID, sequence and length of the sequence.

Ordinal encoding DNA sequence data

In [2]:
```python
import numpy as np
import re
def string_to_array(seq_string):
   seq_string = seq_string.lower()
   seq_string = re.sub('[^acgt]', 'n', seq_string)
   seq_string = np.array(list(seq_string))
   return seq_string

from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()
label_encoder.fit(np.array(['a','c','g','t','z']))
```

Out[2]: `LabelEncoder()`

And here is a function to encode a DNA sequence string as an ordinal vector. It returns a NumPy array with A=0.25, C=0.50, G=0.75, T=1.00, n=0.00.

In [3]:
```python
def ordinal_encoder(my_array):
    integer_encoded = label_encoder.transform(my_array)
    float_encoded = integer_encoded.astype(float)
    float_encoded[float_encoded == 0] = 0.25 # A
    float_encoded[float_encoded == 1] = 0.50 # C
    float_encoded[float_encoded == 2] = 0.75 # G
    float_encoded[float_encoded == 3] = 1.00 # T
    float_encoded[float_encoded == 4] = 0.00 # anything else, lets say n
    return float_encoded


seq_test = 'TTCAGCCAGTG'
ordinal_encoder(string_to_array(seq_test))
```

Out[3]:
```
array([1.  , 1.  , 0.5 , 0.25, 0.75, 0.5 , 0.5 , 0.25, 0.75, 1.  , 0.75])
```

One-hot encoding DNA Sequence

In [4]:
```python
from sklearn.preprocessing import OneHotEncoder
def one_hot_encoder(seq_string):
    int_encoded = label_encoder.transform(seq_string)
    onehot_encoder = OneHotEncoder(sparse=False, dtype=int)
    int_encoded = int_encoded.reshape(len(int_encoded), 1)
    onehot_encoded = onehot_encoder.fit_transform(int_encoded)
    onehot_encoded = np.delete(onehot_encoded, -1, 1)
    return onehot_encoded


seq_test = 'GAATTCTCGAA'
one_hot_encoder(string_to_array(seq_test))
```

Out[4]:
```
array([[0, 0, 1],
       [1, 0, 0],
       [1, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 1, 0],
       [0, 0, 0],
       [0, 1, 0],
       [0, 0, 1],
       [1, 0, 0],
       [1, 0, 0]])
```

DNA sequence as a "language", known as k-mer counting

In [5]:
```python
def Kmers_funct(seq, size):
    return [seq[x:x+size].lower() for x in range(len(seq) - size + 1)]


mySeq = 'GTGCCCAGGTTCAGTGAGTGACACAGGCAG'
Kmers_funct(mySeq, size=7)
```

Out[5]:
```
['gtgccca',
 'tgcccag',
 'gcccagg',
 'cccaggt',
 'ccaggtt',
 'caggttc',
 'aggttca',
 'ggttcag',
 'gttcagt',
 'ttcagtg',
 'tcagtga',
 'cagtgag',
 'agtgagt',
 'gtgagtg',
 'tgagtga',
 'gagtgac',
 'agtgaca',
 'gtgacac',
 'tgacaca',
 'gacacag',
 'acacagg',
 'cacaggc',
 'acaggca',
 'caggcag']
```

It returns a list of k-mer "words." You can then join the "words" into a "sentence", then apply your favorite natural language processing methods on the "sentences" as you normally would.

In [6]:
```python
words = Kmers_funct(mySeq, size=6)
joined_sentence = ' '.join(words)
joined_sentence
```

Out[6]:
```
'gtgccc tgccca gcccag cccagg ccaggt caggtt aggttc ggttca gttcag ttcagt tcagtg cagtga a
gtgag gtgagt tgagtg gagtga agtgac gtgaca tgacac gacaca acacag cacagg acaggc caggca agg
cag'
```

In [7]:
```python
mySeq1 = 'TCTCACACATGTGCCAATCACTGTCACCC'
mySeq2 = 'GTGCCCAGGTTCAGTGAGTGACACAGGCAG'
sentence1 = ' '.join(Kmers_funct(mySeq1, size=6))
sentence2 = ' '.join(Kmers_funct(mySeq2, size=6))
```

In [8]:
```python
#Creating the Bag of Words model:
from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer()
X = cv.fit_transform([joined_sentence, sentence1, sentence2]).toarray()
X
```

Out[8]:
```
array([[0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0,
        1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1,
        0, 1, 0, 0, 1],
       [1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1,
        0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0,
        1, 0, 1, 1, 0],
       [0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0,
        1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1,
        0, 1, 0, 0, 1]], dtype=int64)
```

Here comes machine learning...

In [9]:
```python
import numpy as np
import pandas as pd
import os
import matplotlib.pyplot as plt
%matplotlib inline
for dirname, _, filenames in os.walk('/ashri/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

Load human DNA data

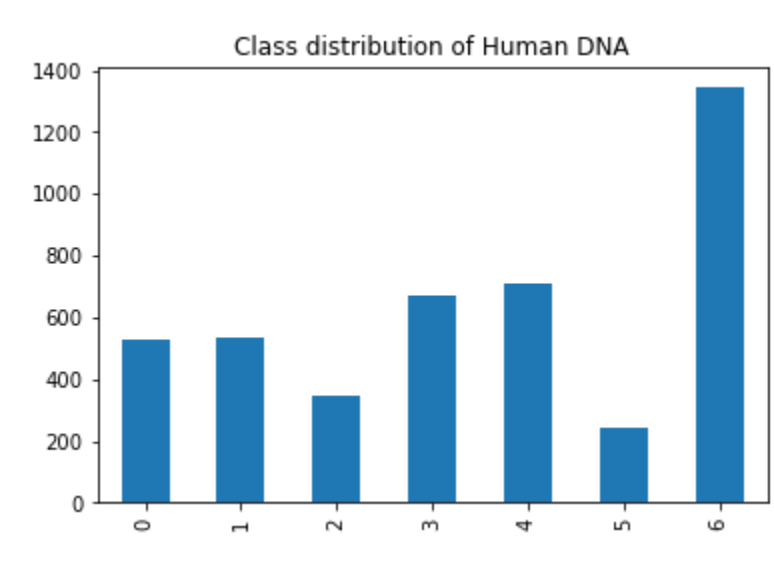
In [10]:
```python
human_dna = pd.read_table('C:\\Users\\ashri\\Downloads\\IBS3EndSem Project\\human.txt
human_dna.head()
```

Out[10]:

	sequence	class
0	ATGCCCCAACTAAATACTACCGTATGGCCCACCATAATTACCCCCA...	4
1	ATGAACGAAAATCTGTTCGCTTCATTCATTGCCCCCACAATCCTAG...	4
2	ATGTGTGGCATTTGGGCGCTGTTTGGCAGTGATGATTGCCTTTCTG...	3
3	ATGTGTGGCATTTGGGCGCTGTTTGGCAGTGATGATTGCCTTTCTG...	3
4	ATGCAACAGCATTTTGAATTTGAATACCAGACCAAAGTGGATGGTG...	3

In [11]:
```python
human_dna['class'].value_counts().sort_index().plot.bar()
plt.title("Class distribution of Human DNA")
```

Out[11]: `Text(0.5, 1.0, 'Class distribution of Human DNA')`



Load Chimpanzee DNA data

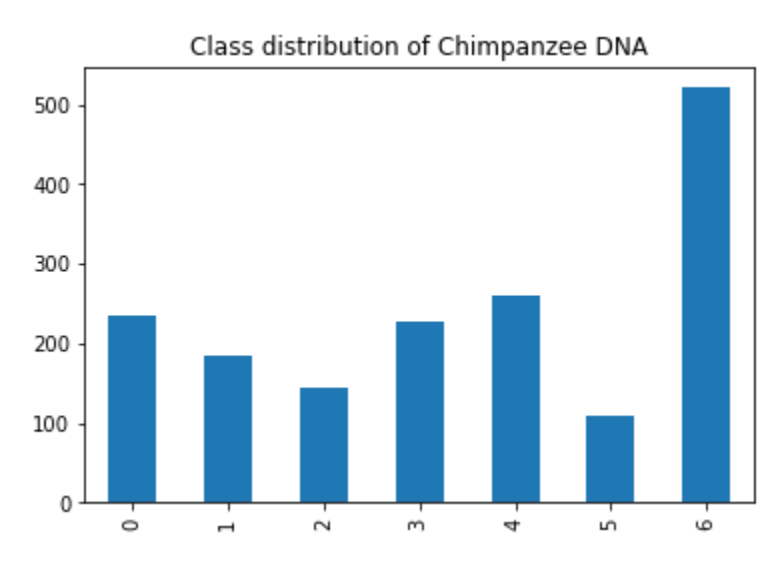
In [12]:
```python
chimp_dna = pd.read_table('C:\\Users\\ashri\\Downloads\\IBS3EndSem Project\\chimpanzee
chimp_dna.head()
```

Out[12]:

	sequence	class
0	ATGCCCCAACTAAATACCGCCGTATGACCCACCATAATTACCCCCA...	4
1	ATGAACGAAAATCTATTCGCTTCATTCGCTGCCCCCACAATCCTAG...	4
2	ATGGCCTCGCGCTGGTGGCGGTGGCGACGCGGCTGCTCCTGGAGGC...	4
3	ATGGCCTCGCGCTGGTGGCGGTGGCGACGCGGCTGCTCCTGGAGGC...	4
4	ATGGGCAGCGCCAGCCCGGGTCTGAGCAGCGTGTCCCCCAGCCACC...	6

In [13]:
```python
chimp_dna['class'].value_counts().sort_index().plot.bar()
plt.title("Class distribution of Chimpanzee DNA")
```

Out[13]: `Text(0.5, 1.0, 'Class distribution of Chimpanzee DNA')`



Load Dog DNA data

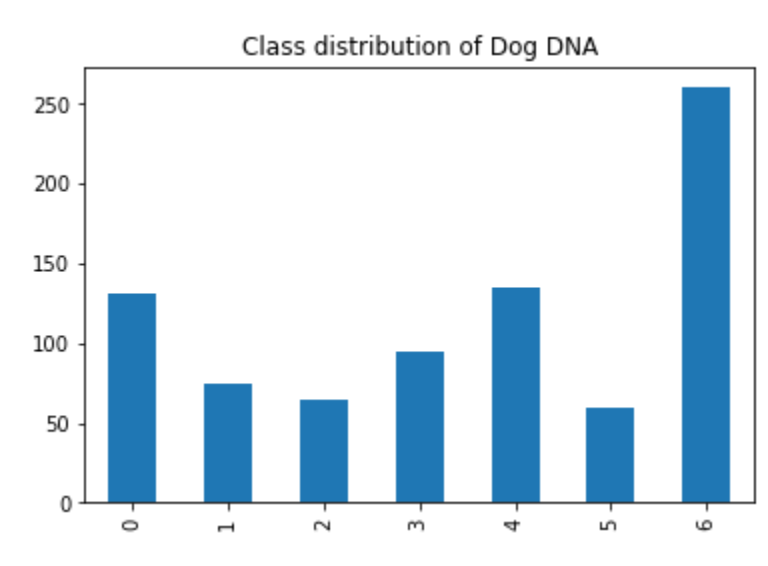
In [14]:
```python
dog_dna = pd.read_table('C:\\Users\\ashri\\Downloads\\IBS3EndSem Project\\dog.txt')
dog_dna.head()
```

Out[14]:

	sequence	class
0	ATGCCACAGCTAGATACATCCACCTGATTTATTATAATCTTTTCAA...	4
1	ATGAACGAAAATCTATTCGCTTCTTTCGCTGCCCCCTCAATAATAG...	4
2	ATGGAAACACCCTTCTACGGCGATGAGGCGCTGAGCGGCCTGGGCG...	6
3	ATGTGCACTAAAATGGAACAGCCCTTCTACCACGACGACTCATACG...	6
4	ATGAGCCGGCAGCTAAACAGAAGCCAGAACTGCTCCTTCAGTGACG...	0

In [15]:
```python
dog_dna['class'].value_counts().sort_index().plot.bar()
plt.title("Class distribution of Dog DNA")
```

Out[15]: `Text(0.5, 1.0, 'Class distribution of Dog DNA')`



Here are the definitions for each of the 7 classes and how many there are in the human training data:

Gene family	Number	Class label
G protein coupled receptors	531	0
Tyrosine kinase	534	1
Tyrosine phosphatase	349	2
Synthetase	672	3
Synthase	711	4
Ion channel	240	5
Transcription factor	1343	6

In [16]:
```python
def Kmers_funct(seq, size=6):
    return [seq[x:x+size].lower() for x in range(len(seq) - size + 1)]


human_dna['words'] = human_dna.apply(lambda x: Kmers_funct(x['sequence']), axis=1)
human_dna = human_dna.drop('sequence', axis=1)

chimp_dna['words'] = chimp_dna.apply(lambda x: Kmers_funct(x['sequence']), axis=1)
chimp_dna = chimp_dna.drop('sequence', axis=1)

dog_dna['words'] = dog_dna.apply(lambda x: Kmers_funct(x['sequence']), axis=1)
dog_dna = dog_dna.drop('sequence', axis=1)
```

In [17]:
```python
human_dna.head()
```

Out[17]:

	class	words
0	4	[atgccc, tgcccc, gcccca, ccccaa, cccaac, ccaac...
1	4	[atgaac, tgaacg, gaacga, aacgaa, acgaaa, cgaaa...
2	3	[atgtgt, tgtgtg, gtgtgg, tgtggc, gtggca, tggca...
3	3	[atgtgt, tgtgtg, gtgtgg, tgtggc, gtggca, tggca...
4	3	[atgcaa, tgcaac, gcaaca, caacag, aacagc, acagc...

We need to now convert the lists of k-mers for each gene into string sentences of words that can be used to create the Bag of Words model. We will make a target variable y to hold the class labels.

In [18]:
```python
human_texts = list(human_dna['words'])
for item in range(len(human_texts)):
    human_texts[item] = ' '.join(human_texts[item])

y_human = human_dna.iloc[:, 0].values
```

Now let's do the same for chimp and dog.

In [19]:
```python
chimp_texts = list(chimp_dna['words'])
for item in range(len(chimp_texts)):
    chimp_texts[item] = ' '.join(chimp_texts[item])

y_chim = chimp_dna.iloc[:, 0].values

dog_texts = list(dog_dna['words'])
for item in range(len(dog_texts)):
    dog_texts[item] = ' '.join(dog_texts[item])

y_dog = dog_dna.iloc[:, 0].values
```

In [20]:
```python
y_human
```

Out[20]: `array([4, 4, 3, ..., 6, 6, 6], dtype=int64)`

So the target variable contains an array of class values.

In [21]:
```python
from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer(ngram_range=(4,4)) #The n-gram size of 4 is determined by testing
X = cv.fit_transform(human_texts)
X_chimp = cv.transform(chimp_texts)
X_dog = cv.transform(dog_texts)
```

In [22]:
```python
print(X.shape)
print(X_chimp.shape)
print(X_dog.shape)
```

```
(4380, 232414)
(1682, 232414)
(820, 232414)
```

In [23]:
```python
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y_human,
                                                    test_size = 0.20,
                                                    random_state=42)
```

We will create a multinomial naive Bayes classifier.

In [24]:
```python
from sklearn.naive_bayes import MultinomialNB
classifier = MultinomialNB(alpha=0.1)
classifier.fit(X_train, y_train)
```

Out[24]: `MultinomialNB(alpha=0.1)`

Now let's make predictions on the human hold out test set and see how it performes on unseen data.

In [25]:
```python
y_pred = classifier.predict(X_test)
```

In [26]:
```python
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score
print("Confusion matrix for predictions on human test DNA sequence\n")
print(pd.crosstab(pd.Series(y_test, name='Actual'), pd.Series(y_pred, name='Predicted
def get_metrics(y_test, y_predicted):
    accuracy = accuracy_score(y_test, y_predicted)
    precision = precision_score(y_test, y_predicted, average='weighted')
    recall = recall_score(y_test, y_predicted, average='weighted')
    f1 = f1_score(y_test, y_predicted, average='weighted')
    return accuracy, precision, recall, f1
accuracy, precision, recall, f1 = get_metrics(y_test, y_pred)
print("accuracy = %.3f \nprecision = %.3f \nrecall = %.3f \nf1 = %.3f" % (accuracy, p
```

```
Confusion matrix for predictions on human test DNA sequence

Predicted   0    1   2    3    4   5    6
Actual
0          99    0   0    0    1   0    2
1           0  104   0    0    0   0    2
2           0    0  78    0    0   0    0
3           0    0   0  124    0   0    1
4           1    0   0    0  143   0    5
5           0    0   0    0    0  51    0
6           1    0   0    1    0   0  263
accuracy = 0.984
precision = 0.984
recall = 0.984
f1 = 0.984
```

Let us now do predictions on Chimp test DNA sequence.

In [27]:
```python
y_pred_chimp = classifier.predict(X_chimp)
```

In [28]:
```python
print("Confusion matrix for predictions on Chimpanzee test DNA sequence\n")
print(pd.crosstab(pd.Series(y_chim, name='Actual'), pd.Series(y_pred_chimp, name='Pre
accuracy, precision, recall, f1 = get_metrics(y_chim, y_pred_chimp)
print("accuracy = %.3f \nprecision = %.3f \nrecall = %.3f \nf1 = %.3f" % (accuracy, p
```

```
Confusion matrix for predictions on Chimpanzee test DNA sequence

Predicted    0    1    2    3    4    5    6
Actual
0          232    0    0    0    0    0    2
1            0  184    0    0    0    0    1
2            0    0  144    0    0    0    0
3            0    0    0  227    0    0    1
4            2    0    0    0  254    0    5
5            0    0    0    0    0  109    0
6            0    0    0    0    0    0  521
accuracy = 0.993
precision = 0.994
recall = 0.993
f1 = 0.993
```

Let us now do predictions on Dog test DNA sequence.

In [29]:
```python
y_pred_dog = classifier.predict(X_dog)
```

In [30]:
```python
print("Confusion matrix for predictions on Dog test DNA sequence\n")
print(pd.crosstab(pd.Series(y_dog, name='Actual'), pd.Series(y_pred_dog, name='Predict
accuracy, precision, recall, f1 = get_metrics(y_dog, y_pred_dog)
print("accuracy = %.3f \nprecision = %.3f \nrecall = %.3f \nf1 = %.3f" % (accuracy, p
```

```
Confusion matrix for predictions on Dog test DNA sequence

Predicted    0   1   2   3    4   5    6
Actual
0          127   0   0   0    0   0    4
1            0  63   0   0    1   0   11
2            0   0  49   0    1   0   14
3            1   0   0  81    2   0   11
4            4   0   0   1  126   0    4
5            4   0   0   0    1  53    2
6            0   0   0   0    0   0  260
accuracy = 0.926
precision = 0.934
recall = 0.926
f1 = 0.925
```

----------------------THANK YOU----------------------