# Binary Splice: Identification of a DNA sequence with Splice Junction

Suchith Umesh Shetty(HIUH69)

[1]Faculty of Informatics, Department of Data Science and Engineering, ELTE-Eötvös Loránd University, Pázmány Péter sétány 1117, Budapest, Hungary
suchith.shetty19792@gmail.com

**Abstract.** Splice junctions are points on a DNA sequence at which `superfluous' DNA is removed during the process of protein creation in higher organisms. Given a sequence of DNA, this project aims to determine if there exists a splice junction. Results can then be used to further examine whether the junction, if exists, is of "intron->extron" type or "extron->intron" type. After exploring and processing the raw data, the classification task was executed using several ML models and their performance is compared and presented in this paper.

**Keywords:** DNA sequence, Splice-site detection, classification.

## 1 Introduction

### 1.1 Domain context and background

All living organisms have their genetic instructions for evolution encoded in DNA (Deoxyribonucleic Acid). Proteins, which are essential components of all living beings, are produced in our cells according the genetic sequence information that is encoded in our DNA [1]. This protein coding is a two stage process called gene Expression. Stage 1 is called Transcription phase, wherein a mRNA (messenger Ribonucleic Acid) is synthesized, which is subsequently translated into a protein in the second stage called Translation phase.

Any error is either of the 2 phases of gene Expression can lead to genetic disorders. Therefore, ever since the discovery of DNA, there has been growing interest towards understanding and recognizing gene sequence patterns which can be of substantial help in finding a cure to wide range of genetic disorders [2].

### 1.2 Problem scope and objective

One such class of problems in the DNA sequences is the identification of splice junctions, particularly for organisms of class Eukaryotes. Eukaryotes genes are composed of alternated segments of exons and introns. Exons correspond to regions that are translated into proteins, and introns to regions that do not code for proteins. Transla-

tion in eukaryote organisms has then an additional step, where introns are spliced out from the mRNA molecule. *Splice junctions* are the boundary points where splicing occurs.

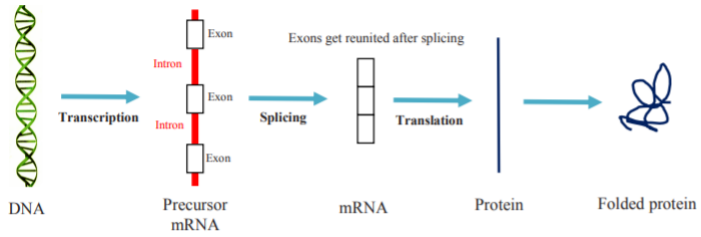



**Fig 1. RNA synthesis and translation into proteins [3]**

The splice junction recognition problem involves identifying if a specified sequence has a splice site or not, and its type (exon-intron or intron-exon)[4]. The scope of this project is limited to the first part of the problem, i.e., classifying whether a given gene sequence has a splice site or not.

### 1.3 Raw data and metadata information

Raw data analyzed in this project was downloaded from:
https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary/splice

Metadata information for the above raw data was obtained from:
https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#splice

Here is the basic metadata information available in the above site:

- # of classes: 2
- # of data: 1,000 / 2,175 (testing)
- # of features: 60

**Table1: Metadata information**

| Attribute | Description |
|---|---|
| 1 | +1 for splice-site and -1 for no splice-site |
| 2-61 | The remaining 60 fields are the sequence, starting at position -30 and ending at position +30. |

# 2 Data analysis and modeling

## 2.1 Reading the data

Data from the above site was downloaded, placed in a local folder and read into python notebook using the below code:

```
data = pd.read_csv('..\\rawdata\splice.txt',delimiter=" ",header=None,prefix='X')
data_scaled = pd.read_csv('..\\rawdata\splice_scale.txt',delimiter=" ",header=None,prefix='X')
data_test = pd.read_csv('..\\rawdata\splice_t.txt',delimiter=" ",header=None,prefix='X')

df = data.copy()
dfs = data_scaled.copy()
dft = data_test.copy()
```

```
df.head()
```

| | X0 | X1 | X2 | X3 | X4 | X5 | X6 | X7 | X8 | X9 | ... | X52 | X53 | X54 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1:2.000000 | 2:1.000000 | 3:2.000000 | 4:1.000000 | 5:2.000000 | 6:1.000000 | 7:2.000000 | 8:3.000000 | 9:3.000000 | ... | 52:3.000000 | 53:4.000000 | 54:2.000000 | 5 |
| 1 | 1 | 1:1.000000 | 2:3.000000 | 3:3.000000 | 4:4.000000 | 5:2.000000 | 6:1.000000 | 7:2.000000 | 8:2.000000 | 9:4.000000 | ... | 52:3.000000 | 53:4.000000 | 54:4.000000 | 5 |
| 2 | 1 | 1:1.000000 | 2:4.000000 | 3:1.000000 | 4:4.000000 | 5:4.000000 | 6:4.000000 | 7:4.000000 | 8:1.000000 | 9:1.000000 | ... | 52:1.000000 | 53:1.000000 | 54:4.000000 | 5 |
| 3 | -1 | 1:1.000000 | 2:1.000000 | 3:4.000000 | 4:1.000000 | 5:1.000000 | 6:3.000000 | 7:3.000000 | 8:4.000000 | 9:4.000000 | ... | 52:3.000000 | 53:4.000000 | 54:1.000000 | 5 |
| 4 | -1 | 1:4.000000 | 2:3.000000 | 3:4.000000 | 4:4.000000 | 5:3.000000 | 6:1.000000 | 7:1.000000 | 8:4.000000 | 9:4.000000 | ... | 52:3.000000 | 53:2.000000 | 54:1.000000 | 5 |

5 rows × 62 columns

## 2.2 Data exploration and preprocessing

Taking the brief look at the data reveals the following:

- Training and test data have same structure and are available in 2 separate files
- Column 1 has the target variable
- Column 2 to 61 has the sequence in the format "<Col No>:<Seq value>"
- Column 62 is dummy with empty values

To account for the above observations, the following data cleansing steps were performed:

```
def datacleanse(df):
    df.drop("X61",axis=1,inplace=True)
    for i in range(1,61):
        df.iloc[:,i]=[float(x.split(":")[1]) for x in df.iloc[:,i]]

datacleanse(df)
datacleanse(dft)
```

```
df_total = pd.concat([df,dft]).drop_duplicates().reset_index(drop=True)
df_total.rename(columns={'X0':'Class'},inplace=True)
```

After the cleaning process:

```
df_total.head()
```

| | Class | X1 | X2 | X3 | X4 | X5 | X6 | X7 | X8 | X9 | ... | X51 | X52 | X53 | X54 | X55 | X56 | X57 | X58 | X59 | X60 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 2.0 | 1.0 | 2.0 | 1.0 | 2.0 | 1.0 | 2.0 | 3.0 | 3.0 | ... | 2.0 | 3.0 | 4.0 | 2.0 | 2.0 | 2.0 | 2.0 | 1.0 | 3.0 | 4.0 |
| **1** | 1 | 1.0 | 3.0 | 3.0 | 4.0 | 2.0 | 1.0 | 2.0 | 2.0 | 4.0 | ... | 4.0 | 3.0 | 4.0 | 4.0 | 4.0 | 1.0 | 3.0 | 4.0 | 1.0 | 1.0 |
| **2** | 1 | 1.0 | 4.0 | 1.0 | 4.0 | 4.0 | 4.0 | 4.0 | 1.0 | 1.0 | ... | 2.0 | 1.0 | 1.0 | 4.0 | 2.0 | 2.0 | 4.0 | 4.0 | 4.0 | 2.0 |
| **3** | -1 | 1.0 | 1.0 | 4.0 | 1.0 | 1.0 | 3.0 | 3.0 | 4.0 | 4.0 | ... | 3.0 | 3.0 | 4.0 | 1.0 | 3.0 | 3.0 | 4.0 | 2.0 | 2.0 | 1.0 |
| **4** | -1 | 4.0 | 3.0 | 4.0 | 4.0 | 3.0 | 1.0 | 1.0 | 4.0 | 4.0 | ... | 1.0 | 3.0 | 2.0 | 1.0 | 4.0 | 2.0 | 1.0 | 1.0 | 4.0 | 4.0 |

5 rows × 61 columns

Class variable is binary as described in the metadata and attributes (i.e., sequence values) seems to be categorical with 4 unique values (1,2,3 and 4 corresponding to A,T,G and C components that make up a DNA sequence [6]) and hence nominal.

Code to get the frequency count by each attribute:

```
#to get freq count
series = []
for name in df_total:
    series.append(df_total[name].value_counts())

info = pd.DataFrame(series)
freq_info = info.transpose()
freq_info
```

| | Class | X1 | X2 | X3 | X4 | X5 | X6 | X7 | X8 | X9 | ... | X51 | X52 | X53 | X54 | X55 | X56 | X57 | X58 | X59 | X60 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **-1.0** | 1344.0 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | ... | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **1.0** | 1647.0 | 698.0 | 735.0 | 665.0 | 716.0 | 760.0 | 656.0 | 738.0 | 738.0 | 703.0 | ... | 659.0 | 762.0 | 685.0 | 656.0 | 753.0 | 715.0 | 695.0 | 682.0 | 734.0 | 679.0 |
| **2.0** | NaN | 821.0 | 748.0 | 797.0 | 751.0 | 691.0 | 774.0 | 728.0 | 696.0 | 708.0 | ... | 860.0 | 826.0 | 809.0 | 864.0 | 811.0 | 779.0 | 853.0 | 808.0 | 744.0 | 875.0 |
| **3.0** | NaN | 779.0 | 795.0 | 817.0 | 816.0 | 803.0 | 842.0 | 804.0 | 820.0 | 837.0 | ... | 804.0 | 742.0 | 779.0 | 764.0 | 772.0 | 771.0 | 750.0 | 801.0 | 806.0 | 745.0 |
| **4.0** | NaN | 693.0 | 713.0 | 712.0 | 708.0 | 737.0 | 719.0 | 721.0 | 737.0 | 743.0 | ... | 668.0 | 661.0 | 718.0 | 707.0 | 655.0 | 726.0 | 693.0 | 700.0 | 707.0 | 692.0 |

5 rows × 61 columns

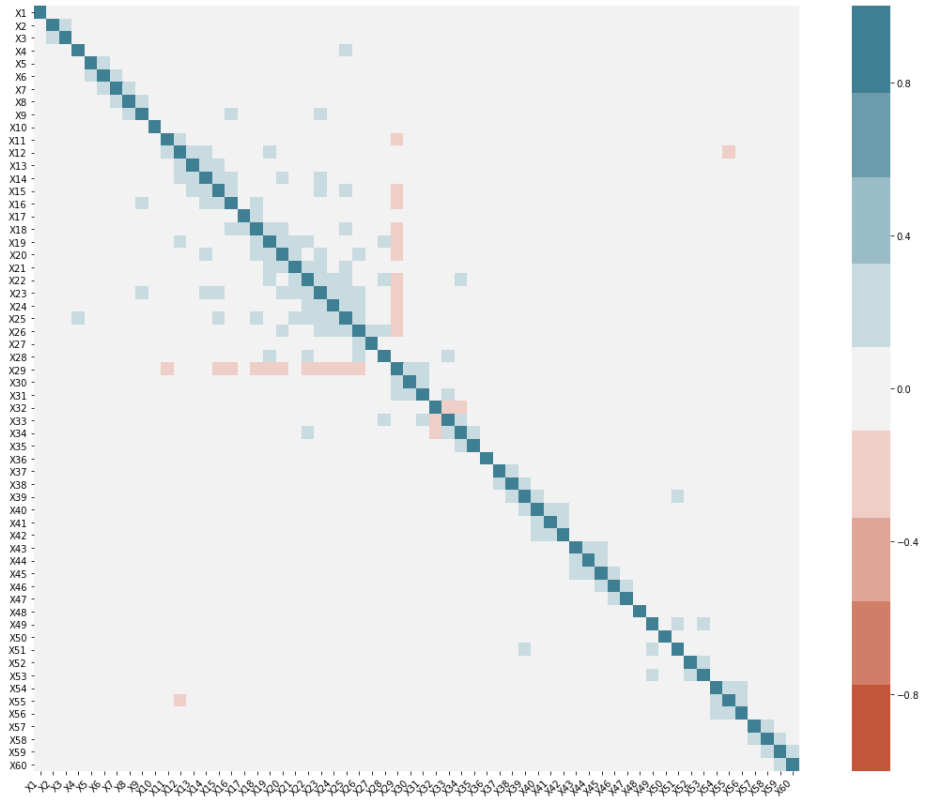
```
def norm(df):
    return (df/df.sum())
freq_info = freq_info.apply(norm)
freq_info
```

| | Class | X1 | X2 | X3 | X4 | X5 | X6 | X7 | X8 | X9 | ... | X51 | X52 | X53 | X54 | X |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **-1.0** | 0.449348 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | ... | NaN | NaN | NaN | NaN | N |
| **1.0** | 0.550652 | 0.233367 | 0.245737 | 0.222334 | 0.239385 | 0.254096 | 0.219325 | 0.246740 | 0.246740 | 0.235038 | ... | 0.220328 | 0.254764 | 0.229020 | 0.219325 | 0.251 |
| **2.0** | NaN | 0.274490 | 0.250084 | 0.266466 | 0.251087 | 0.231026 | 0.258776 | 0.243397 | 0.232698 | 0.236710 | ... | 0.287529 | 0.276162 | 0.270478 | 0.288867 | 0.271 |
| **3.0** | NaN | 0.260448 | 0.265797 | 0.273153 | 0.272818 | 0.268472 | 0.281511 | 0.268806 | 0.274156 | 0.279840 | ... | 0.268806 | 0.248078 | 0.260448 | 0.255433 | 0.258 |
| **4.0** | NaN | 0.231695 | 0.238382 | 0.238047 | 0.236710 | 0.246406 | 0.240388 | 0.241057 | 0.246406 | 0.248412 | ... | 0.223337 | 0.220996 | 0.240053 | 0.236376 | 0.218 |

- There is no class imbalance (45% to 55% split)
- There are no missing values
- Attributes distribution is about uniform in the sequence

On plotting the correlation between the attributes in the sequence, there didn't seem to be any significant correlation worth pursuing:

```
corr = df.iloc[:,1:].corr()
ax = sns.heatmap(
    corr,
    vmin=-1, vmax=1, center=0,
    cmap=sns.diverging_palette(20,220,n=9),
    square=True
)
ax.set_xticklabels(
    ax.get_xticklabels(),
    rotation=45,
    horizontalalignment='right'
);
```



### 2.3 Model building

With the above understanding of the data, I tried to build a simple nearest neighbor classifier on the original training data as shown below:

To determine the optimal value of k to be used in the KNN classifier, the model was iterated over a range of k-values and the one with best accuracy was chosen to test the performance on the test data. The accuracy came out to be highest (~72%) for k=11 and a similar accuracy was observed on original test data as well

```
def train_test_split(data,test_ratio):
    np.random.seed(42)
    shuffled_indices = np.random.permutation(len(data))
    test_size = int(len(data)*test_ratio)
    train_indices = shuffled_indices[:test_size]
    test_indices = shuffled_indices[test_size:]
    return data.iloc[train_indices],data.iloc[test_indices]

X_train,X_test = train_test_split(df.iloc[:,1:61],0.2)
y_train,y_test = train_test_split(df['Class'],0.2)
```

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics

k_range = range(1,20)
scores = {}
scores_list = []
for k in k_range:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train,y_train)
    y_pred = knn.predict(X_test)
    scores[k]=metrics.accuracy_score(y_test,y_pred)
    scores_list.append(metrics.accuracy_score(y_test,y_pred))

plt.plot(k_range,scores_list);
plt.xlabel("k values for KNN")
plt.ylabel("Testing Accuracy")
```

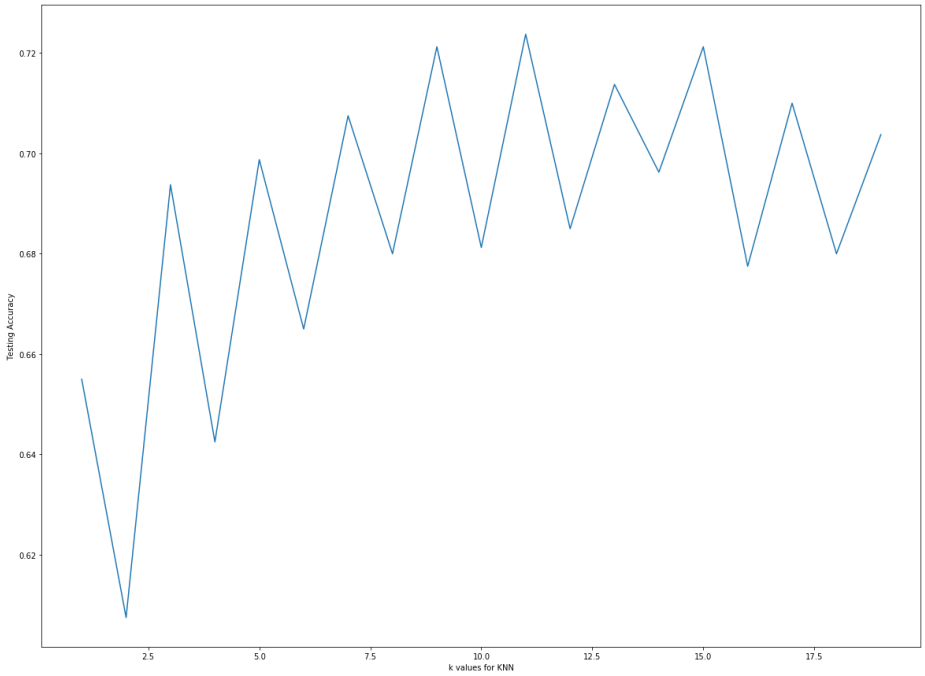



**Fig 2. Plot of accuracy of KNN classifier against k values for the training data**

```
knn = KNeighborsClassifier(n_neighbors=11)
knn.fit(X_train,y_train)
y_pred = knn.predict(X_test2)
print("KNN (k=11): Accuracy(train) = ",metrics.accuracy_score(y_test2,y_pred))
print("Test: ")
print(metrics.classification_report(y_test2, y_pred))
```

```
KNN (k=11): Accuracy(train) =  0.7071264367816092
Test:
              precision    recall  f1-score   support

           0       0.63      0.93      0.75      1044
           1       0.89      0.50      0.64      1131

    accuracy                           0.71      2175
   macro avg       0.76      0.72      0.70      2175
weighted avg       0.77      0.71      0.69      2175
```

The following changes were made to improve on the accuracy of the model:

- Model was trained on the entire data (instead of using just the 1000 records allotted for training), with only 20% data set aside for testing
- Model was trained with 10-fold cross validation approach instead of using a single training data

Results with the revised approach is as shown below:

```
from sklearn import model_selection
X_train,X_test = train_test_split(df_total.iloc[:,1:61],0.2)
y_train,y_test = train_test_split(df_total['Class'],0.2)
model=KNeighborsClassifier(n_neighbors = 11)
kfold = model_selection.KFold(n_splits=10, random_state = 42)
cv_results = model_selection.cross_val_score(model, X_train, y_train, cv=kfold, scoring='accuracy')
msg = "KNN: Train -- %f (%f)" % (cv_results.mean(), cv_results.std())
print(msg)
model.fit(X_train, y_train)
predictions = model.predict(X_test)
print('KNN: Test  -- ',metrics.accuracy_score(y_test, predictions))
print()
print(metrics.classification_report(y_test, predictions))
```

```
KNN: Train -- 0.734153 (0.041225)
KNN: Test  --  0.7271207689093189

              precision    recall  f1-score   support

           0       0.63      0.94      0.76      1083
           1       0.92      0.55      0.69      1310

    accuracy                           0.73      2393
   macro avg       0.78      0.75      0.72      2393
weighted avg       0.79      0.73      0.72      2393
```

Still not as satisfied with the accuracy, other classifiers were considered. Based on the observation that in the attributes are not correlated to each other, Naïve Bayes classifier was tested, since mutually independent features is a primary requirement for this classifier[5]:

```
from sklearn import model_selection
from sklearn.naive_bayes import GaussianNB
X_train,X_test = train_test_split(df_total.iloc[:,1:61],0.2)
y_train,y_test = train_test_split(df_total['Class'],0.2)
model=GaussianNB()
kfold = model_selection.KFold(n_splits=10, random_state = 42)
cv_results = model_selection.cross_val_score(model, X_train, y_train, cv=kfold, scoring='accuracy')
msg = "Naive Bayes: Train -- %f (%f)" % (cv_results.mean(), cv_results.std())
print(msg)
model.fit(X_train, y_train)
predictions = model.predict(X_test)
print('Naive Bayes: Test  -- ',metrics.accuracy_score(y_test, predictions))
print()
print(metrics.classification_report(y_test, predictions))
```

```
Naive Bayes: Train -- 0.864661 (0.034182)
Naive Bayes: Test  --  0.8725449226911827

              precision    recall  f1-score   support

           0       0.88      0.83      0.86      1083
           1       0.87      0.90      0.89      1310

    accuracy                           0.87      2393
   macro avg       0.87      0.87      0.87      2393
weighted avg       0.87      0.87      0.87      2393
```

Accuracy improved quite significantly with Naïve Bayes. Since, SVM is known to be a robust model, we tried to build a SVM classifier using ‘linear’ and ‘poly’ as kernel methods (‘rbf’ and ‘sigmoid’ as kernel didn’t give out good results)

```
def classifier(model,name):
    model=SVC(kernel = 'poly',gamma='auto')
    name="SVC Linear"
    kfold = model_selection.KFold(n_splits=10, random_state = 42)
    cv_results = model_selection.cross_val_score(model, X_train, y_train, cv=kfold, scoring='accuracy')
    msg = name+" : Train -- %f (%f)" % (cv_results.mean(), cv_results.std())
    print(msg)
    model.fit(X_train, y_train)
    predictions = model.predict(X_test)
    print(name,': Test  -- ',metrics.accuracy_score(y_test, predictions))
    print()
    print(metrics.classification_report(y_test, predictions))
```

```
model=SVC(kernel = 'linear',gamma='auto')
name="SVC Linear"
classifier(model,name)
```

```
SVC Linear : Train -- 0.799350 (0.053255)
SVC Linear : Test  --  0.8186376932720435

              precision    recall  f1-score   support

           0       0.81      0.78      0.80      1083
           1       0.82      0.85      0.84      1310

    accuracy                           0.82      2393
   macro avg       0.82      0.82      0.82      2393
weighted avg       0.82      0.82      0.82      2393
```

```
model=SVC(kernel = 'poly',gamma='auto')
name="SVC Poly"
classifier(model,name)
```

```
SVC Poly : Train -- 0.832712 (0.033694)
SVC Poly : Test  --  0.8671124111993314

              precision    recall  f1-score   support

           0       0.84      0.87      0.86      1083
           1       0.89      0.86      0.88      1310

    accuracy                           0.87      2393
   macro avg       0.87      0.87      0.87      2393
weighted avg       0.87      0.87      0.87      2393
```

An ensemble model was also tested and accuracy was further improved:

```
model=AdaBoostClassifier()
name="Ensemble AdaBoost"
classifier(model,name)
```

```
Ensemble AdaBoost : Train -- 0.913079 (0.037941)
Ensemble AdaBoost : Test  --  0.9201838696197242

              precision    recall  f1-score   support

           0       0.91      0.91      0.91      1083
           1       0.93      0.93      0.93      1310

    accuracy                           0.92      2393
   macro avg       0.92      0.92      0.92      2393
weighted avg       0.92      0.92      0.92      2393
```

**Data transformation to improve accuracy:**

To verify if any data transformation would help improve the accuracy of the classifier, all the nominal features were one-hot encoded and the selected models were run again. Below is the code summary and results:

```
for col in df_total.columns:
    df_total[col] = pd.Categorical(df_total[col])
df_ohe = pd.get_dummies(df_total)
df_ohe = df_ohe.drop(columns=['Class_-1'])
df_ohe.rename(columns={'Class_1':'Class'},inplace=True)
df_ohe.head()
```

| | Class | X1_1.0 | X1_2.0 | X1_3.0 | X1_4.0 | X2_1.0 | X2_2.0 | X2_3.0 | X2_4.0 | X3_1.0 | ... | X58_3.0 | X58_4.0 | X59_1.0 | X59_2.0 | X59_3.0 | X59_4.0 | X60_1.0 | X60_ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 1 | 0 | 0 | |
| **1** | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | ... | 0 | 1 | 1 | 0 | 0 | 0 | 1 | |
| **2** | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | ... | 0 | 1 | 0 | 0 | 0 | 1 | 0 | |
| **3** | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 1 | 0 | 0 | 1 | |
| **4** | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 1 | 0 | |

5 rows × 241 columns

```
# define scoring method
scoring = 'accuracy'

# Define models to train
names = ["Nearest Neighbors", "Naive Bayes","SVM Linear", "SVM Poly", "Ensemble AdaBoost"]

classifiers = [
    KNeighborsClassifier(n_neighbors = 9),
    GaussianNB(),
    SVC(kernel = 'linear',gamma='auto'),
    SVC(kernel = 'poly',gamma='auto'),
    AdaBoostClassifier(),
]

models = zip(names, classifiers)

# evaluate each model in turn
results = []
names = []

for name, model in models:
    kfold = model_selection.KFold(n_splits=10, random_state = seed)
    cv_results = model_selection.cross_val_score(model, X_train, y_train, cv=kfold, scoring=scoring)
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)
    model.fit(X_train, y_train)
    predictions = model.predict(X_test)
    print('Test-- ',name,': ',accuracy_score(y_test, predictions))
    print()
    print(classification_report(y_test, predictions))
```

```
Nearest Neighbors: 0.816325 (0.021726)
Test--  Nearest Neighbors :  0.8061497326203209

              precision    recall  f1-score   support

           0       0.69      0.99      0.81       322
           1       0.99      0.67      0.80       426

    accuracy                           0.81       748
   macro avg       0.84      0.83      0.81       748
weighted avg       0.86      0.81      0.80       748

Naive Bayes: 0.910837 (0.016531)
Test--  Naive Bayes :  0.9171122994652406

              precision    recall  f1-score   support

           0       0.91      0.89      0.90       322
           1       0.92      0.93      0.93       426

    accuracy                           0.92       748
   macro avg       0.92      0.91      0.92       748
weighted avg       0.92      0.92      0.92       748

SVM Linear: 0.920202 (0.016089)
Test--  SVM Linear :  0.9411764705882353

              precision    recall  f1-score   support

           0       0.92      0.95      0.93       322
           1       0.96      0.94      0.95       426

    accuracy                           0.94       748
   macro avg       0.94      0.94      0.94       748
weighted avg       0.94      0.94      0.94       748
```

```
SVM Poly: 0.544351 (0.036248)
Test--  SVM Poly :  0.56951871657754

              precision    recall  f1-score   support

           0       0.00      0.00      0.00       322
           1       0.57      1.00      0.73       426

    accuracy                           0.57       748
   macro avg       0.28      0.50      0.36       748
weighted avg       0.32      0.57      0.41       748

C:\ProgramData\Anaconda3\lib\site-packages\sklearn\metr
score are ill-defined and being set to 0.0 in labels wi
  'precision', 'predicted', average, warn_for)
Ensemble AdaBoost: 0.927786 (0.015100)
Test--  Ensemble AdaBoost :  0.9425133689839572

              precision    recall  f1-score   support

           0       0.92      0.95      0.93       322
           1       0.96      0.94      0.95       426

    accuracy                           0.94       748
   macro avg       0.94      0.94      0.94       748
weighted avg       0.94      0.94      0.94       748
```

As seen above, the performed data transformation did improve accuracy for almost all the selected classifiers, except for SVM with kernel as ‘poly’.

# 3 Conclusion

Following the CRISP-DM approach, the classification task of identifying whether a given DNA sequence contains a splice-site or not was performed. Starting with a brief understanding of the domain knowledge and the need for the task at hand, the available dataset was explored and various classifier models were built and compared to achieve highest possible accuracy. Summary table comparing the models can be seen below

| | **Model** | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | **(with original data)** | | | | | | **transformed data (one-hot encoded)** | | | | |
| | **w/o CV** | **with cross-validation** | | | | | | | | | |
| **Metric** | **Nearest Neighbor** | **NN** | **Naïve Bayes** | **SVM Linear** | **SVM Poly** | **Ada-Boost** | **NN** | **Naïve Bayes** | **SVM Linear** | **SVM Poly** | **Ada-Boost** |
| Accuracy (Train) | 71% | 73% | 86% | 80% | 83% | 91% | 81% | 91% | 92% | 54% | 93% |
| Accuracy (Test) | 71% | 73% | 87% | 82% | 87% | 92% | 81% | 92% | 94% | 57% | 94% |
| Precision | 77% | 79% | 87% | 82% | 87% | 92% | 86% | 94% | 94% | 32% | 94% |
| Recall | 71% | 73% | 87% | 82% | 87% | 92% | 81% | 94% | 94% | 57% | 94% |
| F1 measure | 70% | 72% | 87% | 82% | 87% | 92% | 80% | 94% | 94% | 41% | 94% |

# References


1. Lewis, R.: Human Genetics - Concepts and Applications. McGraw Hill (2001).
2. Sadusky T, Newman AJ, and Dibb NJ. Exon Junction Sequences as Cryptic Splice Sites: Implications for Intron Origin. Current Biology, 2004. 14(6): p. 505-509
3. Zaw Zaw Htike and Shoon Lei Win / Procedia Computer Science 23 ( 2013 ) 36 – 43
4. Baldi, P., Brunak, S.: Bioinformatics - The Machine Learning Approach. The MIT Press (1998),
5. Tan,Steinbach, Kumar Introduction to Data Mining 4/18/2004 1 Statistical Inference (By Michael Jordon) l Bayesian perspective.