Predicting the subcellular location of eukaryotic proteins

Said Kassim

CS Dept, UCL, London, WC1E 6BT

Abstract

Motivation: Due to recent advances in genome sequencing, the need for fast and automated methods for analysis of the large amounts of sequence data is apparent. Deciding a function for a protein is a bit challenging where no homology to other proteins with determined functions is available. One thing that could help in deciding the function would be knowing the subcellular location of a protein. With that in mind coming up with an automated method that will assign proteins to a subcellular location would prove very helpful. In this paper one such tool is presented.
Methods: An ensemble of 8 different classifiers is trained to predict the class of an amino acid sequence as belonging to one of 4 subcellular locations i.e. mitochondrial, cytosolic, secreted and nuclear
Results: The classification accuracy on a validation set is estimated to be 68 ± 2 % using cross validation. Other evaluation metrics such as F1 score and AUC are used to determine accuracy of the classifier predictions and will be discussed further.

1 Introduction

Protein sequence classification is an important task for those in the bioinformatics field. Better tools have been continuously created to simplify the task of classifying proteins into their respective families (1). Research is ongoing as it is a non-trivial problem especially with the exponential growth in the amount of sequence data due to advances in sequencing technology.
A small step towards effective protein classification is to determine the subcellular location of a protein which can then be used downstream with other features to predict the family to which an new sequence belongs. This is what we aim to do. The problem is defined as classification of eukaryotic proteins into one of 4 subcellular locations. These are:

1. **Cytosolic** - within the cell itself, but not inside any organelles
2. **Secreted** - proteins which are transported out of a cell
3. **Nuclear** - proteins found / used within the cell's nucleus
4. **Mitochondrial** - proteins transported to the cell's mitochondria

The data used for this problem is in the FASTA format which is described in (2) as a text-based format for representing either nucleotide sequences or peptide sequences. These sequences are made of single-letter codes which are either base pairs or amino acids. A sequence in FASTA format begins with a single-line header followed by lines of sequence data. The header line is distinguished from the sequence data by a greater-than (">") symbol in the first column.

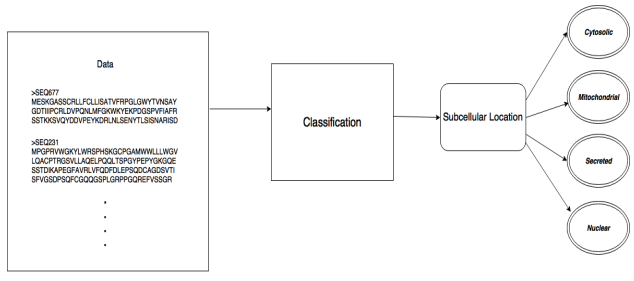

Fig. 1. Classification problem

As you can see in the figure 1, we first have the FASTA format sequence data, which we have to individually classify as one of the 4 subcellular location classes. On the other hand, in (3) they classify into all 14 subcellular locations.
Various different machine learning approaches have been used to tackle the problem of protein classification such as Neural Networks (4), SVM (5), Rough sets (6), Boosting (7) etc. using various combinations of features extracted from the sequence data. I will describe in the next sections the approach I used and the results which I obtained.

2 Approach

The data was made up of 4 separate FASTA files, one for each of the 4 classes. This data was also non homologous meaning there was no need to worry about rebalancing it for the classifier.
After labeling the sequence files I loaded all of them into one primary data structure so I could preprocess them at the same time. Once they were all together I split out the labels which I would use later on to train my classifier. I then performed feature engineering on the data where I came

up with features that I could feed into my machine learning model using some domain knowledge of amino acids. Once I had my features I then normalized them so that I would have a standard normal distribution on each individual feature. This is to ensure the machine learning models will work with the data as some of them assume the data follows a Gausian normal distribution. After normalizing the data I then performed feature selection in order to remain with only those features that help the model and get rid of those that have no predictive power and so may hinder the other features. With my subset of features that I have selected I performed 10 fold cross validation with different classifiers and then with the ensemble of the classifiers. I tuned the classifier hyperparameters as well the ensemble weights in order to optimize my predictions.The experimental results were then evaluated using different classification metrics such as accuracy, F1 and AUC scores. Analysis was done using a confusion matrix of the ensemble to find how it was performing and where the mistakes were being made. Finally, I trained the ensemble on the full training data and predicted the classes of the test data.
Further details of each step of the approach I took will be given in the method section. Figure 2 below shows the full approach taken for the problem.

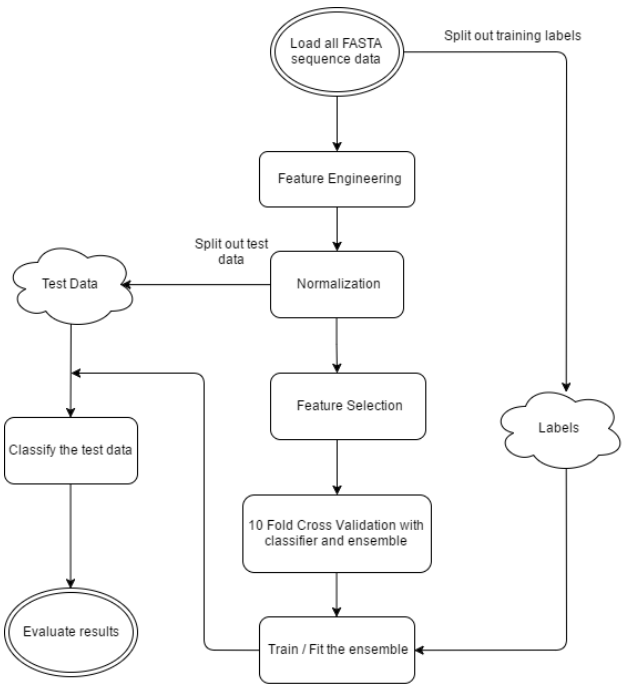

Fig. 2. Approach taken

3 Methods

Each of the steps I took for my classification process will be described in further detail in this section.

3.1 Loading data

With the help of the biopython library, I was able to load the FASTA sequence data and store them into Pandas dataframes. I loaded the 4 different files plus separately and added their respective labels. I then loaded the test data into a dataframe. After I had all the dataframes I concatanated them into 1 large dataframe which I then split out the labels from to be used later for building the classifiers. Figure 3 below shows the dataframe I ended up with. To be able to fit here I took a slice of the first 20 of the 13101 columns and the first 8 of the 9242 rows.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	M	G	Q	Q	V	G	R	V	G	E	A	P	G	L	Q	Q	P	Q	P	R
1	M	A	L	E	P	I	D	Y	T	T	H	S	R	E	I	D	A	E	Y	L
2	M	N	Q	I	E	P	G	V	Q	Y	N	Y	V	Y	D	E	D	E	Y	M
3	M	S	E	E	P	T	P	V	S	G	N	D	K	Q	L	L	N	K	A	W
4	M	G	D	W	M	T	V	T	D	P	G	L	S	S	E	S	K	T	I	S
5	M	V	F	P	A	K	R	F	C	L	V	P	S	M	E	G	V	R	W	A
6	M	E	E	L	I	V	E	L	R	L	F	L	E	L	L	D	H	E	Y	L
7	M	S	G	R	G	N	L	L	S	L	F	N	K	N	A	G	N	M	G	K

Fig. 3. Dataframe of full dataset

3.2 Feature Engineering

For the feature engineering part I leveraged my understanding of amino acids and their properties to come up with 12 different types of features. These features translated to about 1426 dimensions, which I stored in a separate dataframe for use in the machine learning part. The features I came up with are as follows:

3.2.1 Sequence length

The most obvious of the features was to use the sequence length of each sequence in the dataset. The fact that the sequences were of varying sequence meant that this would be a useful signal for the classifier. This added only one dimension to the feature dataframe as the output was only one value for each sequence.

3.2.2 Global amino acid count

The next feature I used was the count of the 20 different types of amino acids across the sequences. This added 20 dimentsions to my feature dataframe, one for each type of amino acid.

3.2.3 Local amino acid count

The local amino acid count is the number of amino acids per window across the sequence from both the front direction and the back. In order to accurately decide the optimal window to count the amino acids in, I slid a window of 10% sequence length from the front increasing the size by 10% and simultaneously slid a similar size window from the end up to the middle where the 2 windows meet for a combination of 25 windows. From my experiments I found the optimal windows to be the first 20% and the last 40% of the sequences. Counting the amino acids within those 2 window sizes gave the best classification accuracy using logistic regression with default parameters. For these experiments I only used this as a feature for the classifier in order to avoid any bias or noise.
This feature also added an extra 20 dimensions. Figure 5 shows the the top 10 results from the experiment.

	Accuracy	Combinations
8	0.631731	[First 20%, Last 40%]
9	0.629924	[First 20%, Last 50%]
23	0.629201	[First 50%, Last 40%]
24	0.628840	[First 50%, Last 50%]
6	0.628840	[First 20%, Last 20%]
7	0.628840	[First 20%, Last 30%]
3	0.628478	[First 10%, Last 40%]
4	0.628478	[First 10%, Last 50%]
14	0.628478	[First 30%, Last 50%]
1	0.628117	[First 10%, Last 20%]

Fig. 4. Best split for local amino acid count

3.2.4 nGram amino acid count

Already having the global amino acid count feature which find the count across the sequence for each single amino acid, I tried combinations of 2 amino acids then 3 amino acids i.e. bigram and trigram amino acid count. This significantly increased the dimensionality of the feature dataframe due to the several different combinations. For the bigram case, the combinations came up to 190 whereas for the trigram they were significantly more at 1140. Unfortunately, this feature wasn't useful as I had anticipated as it lowered the accuracy score using when I tested the features later on. So I dropped the nGram amino acid count feature. It had also made training extra hard as it increased the dimensions by 1330.

3.2.5 Isoelectric Point

The isoelectric point is the pH at which a particular molecule carries no net electrical charge in the statistical mean (8) Due to the conditions of the surrounding environment polypeptides can become more positively or negatively charged. Using biopython, under the SeqUtils package a method of calculating Isoelectric points of sequences can be found and is used here.

3.2.6 Aromaticity

This feature captures the amino acids that include an aromatic ring in their structure. Aromatic rings refer to the rings with resonance bonds that form around cyclic or flat polypeptides and that exhibit more stability than other arrangements. Aromatic molecules are stable and do not break apart easily. Within the 20 amino acids the following exhibit aromaticity:

1. phenylalanine
2. tryptophan
3. tyrosine

Hence the aromaticity measure is the relative frequency of the 3 aromatic amino acids in the sequence.

3.2.7 Gravy

Another feature which relates to amino acid properties is the gravy measure which refers to grand average hydropathy and is calculated by adding the hydropathy value for each residue and dividing by the length of the sequence. Hydropathy refers to the relative scale between hydrophobicity and hydrophilicity, so that the higher the value the more hydrophobic it is and the lower the more hydrophilic it is.

3.2.8 Flexibility

Proteins possess an inherent flexibility which enables them to function through their molecular interactions intra cellular and inter cellular (9). This property of proteins is believed to be found within the amino acid sequences and can therefor be a good feature to classify protein functionality.

3.2.9 Molecular weight

This is a feature that simply captures the molecular weight of the amino acid sequence. This can give hints as to which class a protein may belong.

3.2.10 Instability index

The instability index of a protein refers to whether it will remain stable in a test tube or not, where an index of less than 40 means the protein will be stable in the test tube. If it is greater than this then it is probably not going to be stable. This is used as a feature as well.

3.2.11 Secondary Structure Fraction

This feature refers to the fraction of the sequence that corresponds to alpha helices, beta sheets and turns. Using biopython these can be calculated and the amino acids that belong to each according to their method are as follows:

1. Amino acids in helix: V, I, Y, F, W, L.
2. Amino acids in Turn: N, P, G, S.
3. Amino acids in sheet: E, M, A, L.

3.2.12 Protein scales

More features that are used come from the protein scales such as hydrophobicity, hydrophilicity and surface accessibility which is the surface area of the polypeptide that is accesible by a solvent. It was found to help predict protein function (10)

3.3 Normalization

To make the feature data work for any classifier I normalized the data, using scikit-learn's StandardScaler which centers the data on the mean and then scales it to unit variance therefore making it follow a normal distribution. Normalization is key as it stops some features dominating others by having much larger variances and thus preventing the classifier from learning from other features. I noticed in my experiments that the difference between normalizing and not normalizing was around 4% in accuracy. So that validated the reasoning to perform normalization on the feature dataset. Figure **??** shows the bar chart of the two using classification accuracy scores from logistic regression on default parameters just for a simple comparison.

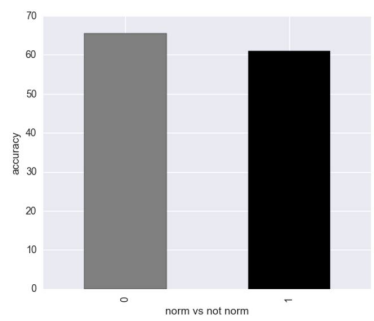

Fig. 5. Normalized vs not normalized accuracy scores

After normalizing the feature dataset I split out the test data which will be used later on after building the classification model.

3.4 Feature Selection

To perform feature selection so as to come up with the best subset of features to fit into my model, I used random forest's feature importance attribute. Figures 6 and 7 below shows the feature importances for the 10 most important features and the least important ones. According to the analysis most of the 96 features seem to be significantly important except the last 2 features, which I then drop and remain with the rest.

	Feature no	Importance
0	65	0.026997
1	5	0.024891
2	10	0.023746
3	1	0.023070
4	37	0.022559
5	79	0.022022
6	70	0.022004
7	31	0.020781
8	7	0.019287
9	0	0.018301

Fig. 6. 10 most important features

	Feature no	Importance
0	51	0.0
1	34	0.0

Fig. 7. The features which will need to be dropped

I also looked at the impact of each feature on the accuracy of the logistic regression to see which features contributed the most as can be seen in figure 8 below.

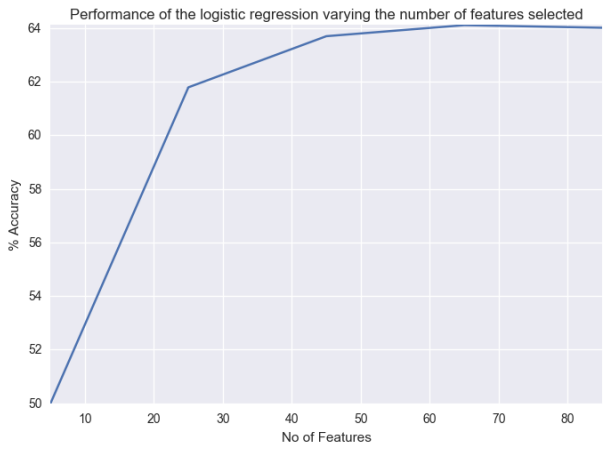

Fig. 8. Accuracy with varying the no of features

3.5 Cross Validation with classifiers

I created 8 different classifiers and then ensembled them. First, I tuned all the parameters using random search with cross validation for 50 iterations. This took nearly 6 hours to finish. After finding the best parameters within the 50 iterations for each of the classification model, I performed 10 fold cross validation on the 8 classifiers as well as the ensemble. For the ensemble I initially set all the weights to be equal. Then after finishing the cross validation I took the accuracy scores which I then scaled with the lowest prediction as 1 and passed these as weights to my ensemble. This ensured the higher scoring classifiers got more weight in the ensemble.

3.6 Fitting the ensemble

After performing cross validation and evaluating the results, the last thing to do is train the ensemble on the full training data and predict the classes of the test data. I used the parameters that I had tuned and the ensemble weights and made the final prediction on the 20 sequences in the test set.

4 Results

The results of the 10 fold cross validation for each of the 8 classifiers as well as the ensemble are shown in figures 10 and 9 below. The best performing models are found to be logistic regression, neural networks (Multilayer Perceptron) and XGBoost with accuracy scores of 65, 66 and 66 % respectively. The combined classifiers produces an accuracy of 68% accuracy with cross validation.

	classifier	mean_accuracy	std_accuracy
0	Logistic Regression	65.094056	2.239781
1	Random Forest	56.073233	3.624478
2	Adaboost	59.347440	1.388817
3	Gradient Boosting	65.473955	1.568209
4	XGboost	66.005349	2.086924
5	SVC	62.068978	1.786908
6	Neural Net	66.080983	2.261822
7	KNN	56.614549	1.657419
8	Ensemble	67.978620	2.255733

Fig. 9. CV scores for each classifier

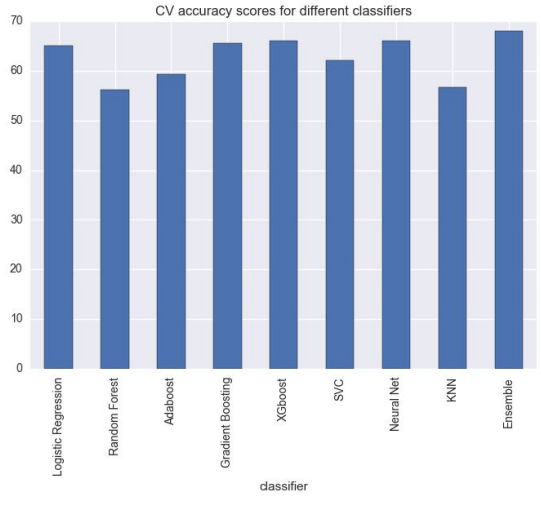

Fig. 10. CV accuracy scores bar graph

The classification accuracy score for the ensemble without cross validation with a 70/30 split of the training data was 70.148%

To better analyse the performance of the ensemble, i calculated the confusion matrix in order to see how well it was predicting for the 4 classes on the predictions for the validation set. The result is shown in figure 11 below.

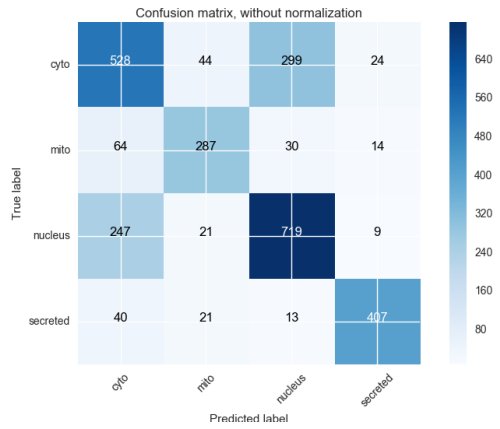

Fig. 11. Ensemble confusion matrix

For the final ensemble model I also evaluated on different metrics as shown in figure 12 below.

	precision	recall	f1-score	support
0	0.60	0.59	0.60	895
1	0.77	0.73	0.75	395
2	0.68	0.72	0.70	996
3	0.90	0.85	0.87	481
avg / total	0.70	0.70	0.70	2767

Fig. 12. Precision, Recall, F1 and Support Metrics Results

Also to analyse which classifiers perform better for each class I performed a ROC AUC analysis for each of the four classes as shown in figures 13141516

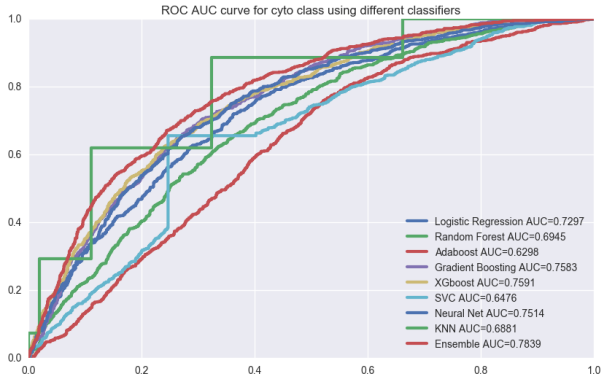

Fig. 13. ROC Curve for Cytosolic

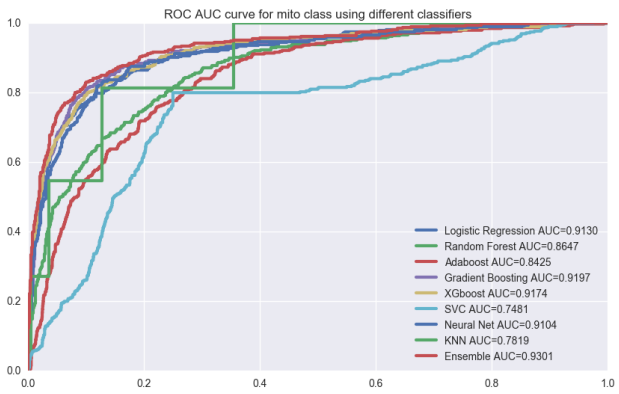

Fig. 14. ROC Curve for Mitochondrial

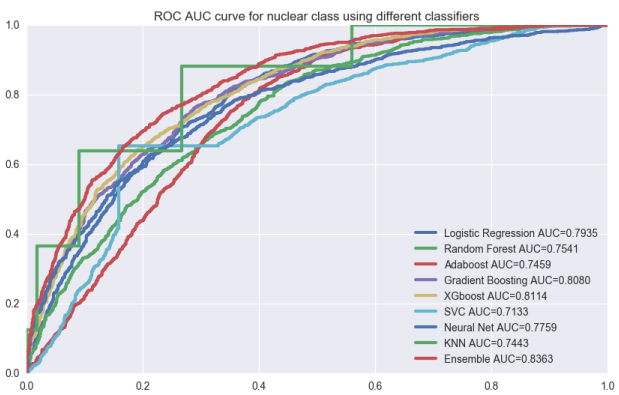

Fig. 15. ROC Curve for Nuclear

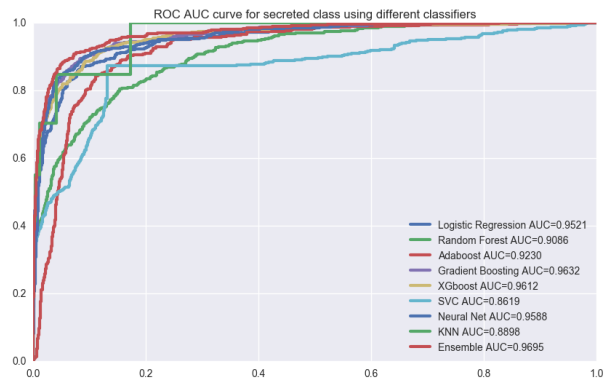

Fig. 16. ROC Curve for Secreted

The final results of the ensemble model on the 'blind' test data is shown in figure 17 below.

	Seq	prediction	measure	confidence
0	SEQ677	cyto	Confidence	38.2%
1	SEQ231	secreted	Confidence	48.2%
2	SEQ871	secreted	Confidence	46.8%
3	SEQ388	nucleus	Confidence	54.8%
4	SEQ122	nucleus	Confidence	57.5%
5	SEQ758	cyto	Confidence	48.3%
6	SEQ333	cyto	Confidence	61.0%
7	SEQ937	cyto	Confidence	69.1%
8	SEQ351	cyto	Confidence	55.4%
9	SEQ202	mito	Confidence	51.7%
10	SEQ608	mito	Confidence	52.6%
11	SEQ402	mito	Confidence	46.1%
12	SEQ433	secreted	Confidence	86.2%
13	SEQ821	secreted	Confidence	78.7%
14	SEQ322	nucleus	Confidence	72.8%
15	SEQ982	nucleus	Confidence	76.4%
16	SEQ951	cyto	Confidence	43.6%
17	SEQ173	cyto	Confidence	57.4%
18	SEQ862	cyto	Confidence	53.1%
19	SEQ224	cyto	Confidence	77.9%

Fig. 17. Final Output Prediction Results on Test data

5 Discussion

My work focused on finding the best features to best describe the amino acid sequence data. I concentrated on feature engineering and found that most of the features I came up with were important except 2. Specifically, the first few features i.e. sequence length, global amino acid count and local amino acid count had the most dramatic effect on the accuracy of the classifier.

The next most important part of the project was to come up with a good classifier. In order to perform an analysis and come up with good predictions, I decided to try different classifiers. Most did well, but the 3 which stood out were Logistic Regression, Gradient Boosting using XGBoost and Neural Networks. In order to get optimal results I did a paramater tuning using random search for all the models I used and after finding these parameters my accuracy improved as compared to the default parameters that come with these classifiers. I also computed the weights to give to the ensemble by taking the relative accuracies of each of the 8 models I used and scaling them with the most accurate getting the highest weight.

Using the ROC AUC graphs I found that different classifiers perform differently for each class where for the Nuclear class Gradient Boosting performed the best with score of greater than 80%. For the cytosolic class Neural Nets and Gradient Boosting performed equally well with score of greater than 75%. For the mitochondrial class, they all performed really well meaning it was easier to classify from the features.Logistic Regression, Neural Nets and Gradient boosting all scored above 90%. For the last class, the secreted class, they all scored highly again with Logistic Regression, Neural Nets and Gradient boosting all scoring above 95%

From the confusion matrix I found that although most of the sequences were being classified correctly as shown the leading diagonal,the ensemble classifier was making the most errors with misclassifying cytosolic and nuclear proteins as each other i.e. the classifier was confusing cytosolic for nuclear and vice versa. This is probably because the proteins in these 2 subcellular locations share certain properties. The functions of the proteins in these 2 locations may be quite similar such as performing cell cycle.

For the final prediction results I used the predicted probabilities from the ensemble classifier as my confidence score.

6 Conclusion

Although 70% is not such a high accuracy score it is quite good given the relatively few number of features that I used. I expect if I had more features which accurately capture the amino acid properties the accuracy of the classifier would improve.
Another idea would be to use better representations of the features which better recognize similarity between features such as Stanford's GloVe or word2vec from Google. Using these better representations would enable the use of state of the art deep learning techniques which would most certainly improve performance as shown by Lee and Nguyen in (11)
More analysis of biological relationships between the different protein families with relation to subcellular location of these proteins would help build better classification models.

7 Appendix

The source code for this project can be found at https://github.com/SaidAbdullahi/Bioinformatics_Coursework Some of the functions I implemeted are shown below. Firstly the feature engineering functions I made:

```
# Feature engineering starts here
# i.e. find relevant features for the sequence data

# 1. Sequence length
sequence_length =
     pd.DataFrame(full_data.count(axis=1),columns=['seq_len'])
sequence_length = sequence_length.astype(float)

# 2. Global amino acid count i.e. no of amino acid per
     sequence
global_count = full_data.T

d = {}
for i in range(len(full_data)):
    series_global = global_count.groupby(i).size()
    d[str(i)]=pd.DataFrame({i:series_global.values},index
         = series_global.index+' global')

global_counts_feats = pd.concat([d[str(i)] for i in
     range(len(d))],axis=1)
global_counts_feats = global_counts_feats.fillna(0)
global_counts_feats = global_counts_feats.T

# 3. Bigram amino counts i.e (AB, AC, AE etc)
aminos = pd.DataFrame(series_global.index)
aminos.columns = ['amino']

bigrams = pd.DataFrame(list(combinations(aminos.amino,
     2)))
bigrams = bigrams.apply(lambda x: ''.join(x), axis=1)
bicnt=[]

for seq in sequences:
    for bi in bigrams:
        bicnt.append(seq.count(bi))

bigramcounts = [bicnt[i:i+len(bigrams)] for i in
     range(0, len(bicnt), len(bigrams))]

bigram_count_df = pd.DataFrame(bigramcounts)
bigram_count_df.columns = bigrams

# 4. Trigram amino counts i.e (ABC, ACE etc)
aminos = pd.DataFrame(series_global.index)
aminos.columns = ['amino']

from itertools import combinations
sequences = cytosequences+mitosequences+
nucsequences+secsequences+blindsequences
trigrams =
     pd.DataFrame(list(combinations(aminos.amino, 3)))
trigrams = trigrams.apply(lambda x: ''.join(x), axis=1)
tricnt=[]

for seq in sequences:
    for tri in trigrams:
        tricnt.append(seq.count(tri))
trigramcounts = [tricnt[i:i+len(trigrams)] for i in
     range(0, len(tricnt), len(trigrams))]

trigram_count_df = pd.DataFrame(trigramcounts)
trigram_count_df.columns = trigrams

# Find the best split for local amino count
aminoFirstCount=[]
aminoLastCount=[]
cnt = np.arange(10,60,10)

for j in cnt:
    for i in range(len(sequences)):
            X=ProteinAnalysis(str(sequences[i][j:]))
            aminoFirstCount.append(X.count_amino_acids())
for j in cnt:
    for i in range(len(sequences)):
            X=ProteinAnalysis(str(sequences[i][:j]))
            aminoLastCount.append(X.count_amino_acids())

aminofirstchunk = [aminoFirstCount[i:i+len(sequences)]
     for i in range(0, len(aminoFirstCount),
     len(sequences))]
aminofirst10 = pd.DataFrame(aminofirstchunk[0])
aminofirst10.columns = [str(cols)+'_first' for cols in
     aminofirst10.columns]
aminofirst20 = pd.DataFrame(aminofirstchunk[1])
aminofirst20.columns = [str(cols)+'_first' for cols in
     aminofirst20.columns]
aminofirst30 = pd.DataFrame(aminofirstchunk[2])
aminofirst30.columns = [str(cols)+'_first' for cols in
     aminofirst30.columns]
aminofirst40 = pd.DataFrame(aminofirstchunk[3])
aminofirst40.columns = [str(cols)+'_first' for cols in
     aminofirst40.columns]
aminofirst50 = pd.DataFrame(aminofirstchunk[4])
aminofirst50.columns = [str(cols)+'_first' for cols in
     aminofirst50.columns]

aminolastchunk = [aminoLastCount[i:i+len(sequences)]
     for i in range(0, len(aminoLastCount),
     len(sequences))]
aminolast10 = pd.DataFrame(aminolastchunk[0])
aminolast10.columns = [str(cols)+'_last' for cols in
     aminolast10.columns]
aminolast20 = pd.DataFrame(aminolastchunk[1])
aminolast20.columns = [str(cols)+'_last' for cols in
     aminolast20.columns]
aminolast30 = pd.DataFrame(aminolastchunk[2])
aminolast30.columns = [str(cols)+'_last' for cols in
     aminolast30.columns]
aminolast40 = pd.DataFrame(aminolastchunk[3])
aminolast40.columns = [str(cols)+'_last' for cols in
     aminolast40.columns]
aminolast50 = pd.DataFrame(aminolastchunk[4])
```

```
aminolast50.columns = [str(cols)+'_last' for cols in
     aminolast50.columns]

acc=[]
feats=[]

for feat_first, featfirst_name in zip([aminofirst10,
     aminofirst20, aminofirst30, aminofirst40,
     aminofirst50],
                    ['First 10%', 'First 20%', 'First
                         30%','First 40%', 'First
                         50%']):
    for feat_last, featlast_name in zip([aminolast10,
         aminolast20, aminolast30, aminolast40,
         aminolast50],
                    ['Last 10%', 'Last 20%', 'Last
                         30%','Last 40%', 'Last 50%']):
        features_df = pd.concat([feat_first,
             feat_last],1)
        labels = labels[:train_idx]
        test_df = features_df[train_idx:]
        features_df = features_df[:train_idx]
        le = preprocessing.LabelEncoder()
        le.fit(labels.values)
        labels_enc=le.transform(labels)
        labels_df =
             pd.DataFrame(labels_enc,columns=['labels'])
        X_train, X_test, y_train, y_test =
             train_test_split(features_df, labels_enc,
             random_state=0, test_size=0.3)
        lr = LogisticRegression()
        y_pred_lr = lr.fit(X_train,
             y_train).predict(X_test)
        acc.append(metrics.accuracy_score(y_test,
             y_pred_lr))
        feats.append([featfirst_name, featlast_name])
results = pd.DataFrame()
results['Accuracy'] = acc
results['Combinations'] = feats


results = results.sort_values(by='Accuracy',
     ascending=False)


# 6. First 50 amino acid count
local_count_first50 =
     global_count.drop(global_count.index[50:])

d = {}
for i in range(len(full_data)):
    series_global =
         local_count_first50.groupby(i).size()
    d[str(i)]=pd.DataFrame({i:series_global.values},index
         = series_global.index+' first50')

first50_counts_feats = pd.concat([d[str(i)] for i in
     range(len(d))],axis=1)
first50_counts_feats = first50_counts_feats.fillna(0)
first50_counts_feats = first50_counts_feats.T

# 7. Last 50 amino acid count
local_count_last50 =
     global_count.drop(global_count.index[:50])

d = {}
for i in range(len(full_data)):
    series_global = local_count_last50.groupby(i).size()
    d[str(i)]=pd.DataFrame({i:series_global.values},index
         = series_global.index+' last50')

last50_counts_feats = pd.concat([d[str(i)] for i in
     range(len(d))],axis=1)
last50_counts_feats = last50_counts_feats.fillna(0)
last50_counts_feats = last50_counts_feats.T

## Biopython Protein Analysis Features
# 8. Isoelectric Point
# 9. Aromaticity
# 10. Secondary Structure Fraction
# 11. Gravy
# 12. Instability Index
# 13. Flexibility
# 14. Amino Percent
# 15. Molecular Weight
# 16. Protein Scale ~ Hydrophobicity
# 17. Protein Scale ~ Hydrophilicity
# 18. Protein Scale ~ Surface accessibility

sequences = cytosequences+mitosequences+nucsequences
+secsequences+blindsequences
isoelectricPt=[]
aromaticity=[]
aminoPercent=[]
secstruct=[]
hydrophob=[]
hydrophil=[]
surface=[]
gravy=[]
molweight=[]
instidx=[]
flex=[]

for seq in sequences:
        X=ProteinAnalysis(str(seq))
        isoelectricPt.append(X.isoelectric_point())
        aromaticity.append(X.aromaticity())
        aminoPercent.append(X.get_amino_acids_percent())
        secstruct.append(X.secondary_structure_fraction())

# These features throw Key & Value Errors due to non
     standard amino acids
# (i.e. out of the 20 standard ones) e.g. X, U etc
        try:
            gravy.append(X.gravy())
            molweight.append(X.molecular_weight())
            instidx.append(X.instability_index())
            flex.append(X.flexibility())
            hydrophob.append(X.protein_scale(ProtParamData.kd,
                 9, 0.4))
            hydrophil.append(X.protein_scale(ProtParamData.hw,
                 9, 0.4))
            surface.append(X.protein_scale(ProtParamData.em,
                 9, 0.4))

        except (KeyError,ValueError):
            gravy.append(0)
            molweight.append(0)
            instidx.append(0)
            flex.append([0,0])
            hydrophob.append([0,0])
            hydrophil.append([0,0])
            surface.append([0,0])

isoelectricPt_df =
     pd.DataFrame(isoelectricPt,columns=['isoelectricPt'])
```

```
aromaticity_df =
     pd.DataFrame(aromaticity,columns=['aromaticity'])
aminoPercent_df = pd.DataFrame()
aminoPercent_df =
     aminoPercent_df.from_dict(aminoPercent)
aminoPercent_df.columns = [str(col) + '%' for col in
     aminoPercent_df.columns]
secstruct_df =
     pd.DataFrame(secstruct,columns=['helix','turn','sheet'])
instidx_df = pd.DataFrame(instidx,
     columns=['instabilityIdx'])
gravy_df = pd.DataFrame(gravy, columns=['gravy'])
molWeight_df = pd.DataFrame(molweight,
     columns=['molWeight'])
flex_df =
     pd.DataFrame(pd.DataFrame(flex).mean(axis=1),
     columns=['flexibility'])
hydrophob_df =
     pd.DataFrame(pd.DataFrame(hydrophob).mean(axis=1),
     columns=['hydrophobicity'])
hydrophil_df =
     pd.DataFrame(pd.DataFrame(hydrophil).mean(axis=1),
     columns=['hydrophilicity'])
surface_df =
     pd.DataFrame(pd.DataFrame(surface).mean(axis=1),
     columns=['surface_accesibility'])
```

Below is my code for building my classifiers and ensemble.

```
# Ensemble of classifiers
X = features_df_scaled
y = labels_enc

hidden_size =
     (int(opt_params_df.NN__hidden_layer_sizes[0].strip('')
     ( ,")),)

# Models with already tuned parameters
clf1 = LogisticRegression(C=opt_params_df.lr__C[0])
clf2 = RandomForestClassifier(n_estimators=int(
opt_params_df.rf__n_estimators[0]),
max_depth=opt_params_df.rf__max_depth[0])
clf3 = AdaBoostClassifier(
n_estimators=int(opt_params_df.ada__n_estimators[0]))
clf4 = GradientBoostingClassifier(n_estimators=
int(opt_params_df.gbr__n_estimators[0]))
clf5 = xgb.XGBClassifier(
max_depth=int(opt_params_df.xg__max_depth[0]),
min_child_weight=int(opt_params_df.xg__min_child_weight[0]))
clf6 = SVC(probability=True,C=opt_params_df.SVC__C[0])
clf7 = MLPClassifier(hidden_layer_sizes=hidden_size,
activation=opt_params_df.NN__activation[0],
                   solver=opt_params_df.NN__solver[0],
                   alpha=opt_params_df.NN__alpha[0])
clf8 = KNeighborsClassifier(
n_neighbors=int(opt_params_df.KNN__n_neighbors))

# weights already tuned
weights = [10.0, 1.0, 4.0, 10.0, 11.0, 7.0, 11.0, 1.0]
eclf = VotingClassifier(estimators=[('lr', clf1),
     ('rf', clf2), ('ada', clf3), ('gbr', clf4),
     ('xgb', clf5),
                                    ('SVC', clf6),
                                         ('Neural', clf7),
                                         ('KNN', clf8)],
                                         voting='soft',
                        weights=weights)


scores_list=[]
# Cross validation performance
for clf, label in zip([clf1, clf2, clf3, clf4, clf5,
     clf6, clf7, clf8, eclf],
                      ['Logistic Regression', 'Random
                           Forest', 'Adaboost','Gradient
                           Boosting',
                       'XGboost', 'SVC', 'Neural
                            Net','KNN', 'Ensemble']):
    scores = cross_val_score(clf, X, y, cv=5,
         scoring='accuracy')
    scores_list.append([label,scores.mean(),scores.std()])
    print("Accuracy: %0.2f (+/- %0.2f) [%s]" %
         (scores.mean(), scores.std(), label))
scores_df = pd.DataFrame(scores_list)
```

References

[1]Bernardes, Juliana S., Jorge H. Fernandez, and Ana Tereza R. Vasconcelos. "Structural descriptor database: a new tool for sequence-based functional site prediction." BMC bioinformatics 9.1 (2008): 492.

[2]http://zhanglab.ccmb.med.umich.edu/FASTA/

[3]Chou, KuoâŁŁChen, and YuâŁŁDong Cai. "Prediction and classification of protein subcellular locationâŁ"sequenceâŁŁorder effect and pseudo amino acid composition." Journal of cellular biochemistry 90.6 (2003): 1250-1260.

[4]Saha, Suprativ, and Rituparna Chaki. "A brief review of data mining application involving protein sequence classification." Advances in Computing and Information Technology. Springer Berlin Heidelberg, 2013. 469-477.

[5]Zhou GP: An intriguing controversy over protein structural class prediction. J Protein Chem 1998, 17: 729âŁ“738. 10.1023/A:1020713915365

[6]Cao YF, Liu S, Zhang LD, Qin J, Wang J, Tang KX: Prediction of protein structural class with Rough Sets. BMC Bioinform 2006., 7:

[7]Feng KY, Cai YD, Chou KC: Boosting classifier for predicting protein domain structural class. Biochem Biophys Res Commun 2005, 334: 213âŁ“217. 10.1016/j.bbrc.2005.06.075

[8]https://en.wikipedia.org/wiki/Isoelectric_point

[9]Teilum, Kaare, Johan G. Olsen, and Birthe B. Kragelund. "Functional aspects of protein flexibility." Cellular and Molecular Life Sciences 66.14 (2009): 2231.

[10]Momen-Roknabadi, A; Sadeghi, M; Pezeshk, H; Marashi, SA (2008). "Impact of residue accessible surface area on the prediction of protein secondary structures". BMC Bioinformatics. 9: 357. doi:10.1186/1471-2105-9-357

[11]https://cs224d.stanford.edu/reports/LeeNguyen.pdf