Machine Learning Report

April 30, 2025

1 Protein Subcellular Localization Prediction: A Machine Learning Approach for Gram-Positive Bacterial Proteins

---

* By Bao, Cherif, and Lucas Liona
* Professor Dehzangi

This is a Jupyter Notebook, this allows us to both show the code/statistics and give our commentary on challenges we faced and how each step impacted our final project

1.1 Project Outline

- Step 1: Read data and extract relevant features (can be done manually with Occurence and Composition)
- Step 2: You will need to prepare the data (properly put them together with labels) and use different ML methods (KNN, SVM, Bayes, ANN, Random Forest).
  - This is a multiclass classification problem
- Step 3: Analyze and Interpret output
  - Independent Test Set
  - K-Fold Cross Validation
  - Accuracy, Precision, Recall, AUC
  - Dicuss Reults and Interprate Output

2 Step 1: Organize Data

[1]:
```
import pandas as pd
import numpy as np
from collections import Counter
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.model_selection import train_test_split, cross_val_score,␣
↪GridSearchCV, StratifiedKFold
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier, VotingClassifier,␣
↪StackingClassifier, AdaBoostClassifier
```

```
from sklearn.naive_bayes import GaussianNB
from sklearn.neural_network import MLPClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.feature_selection import SelectFromModel
from sklearn.metrics import accuracy_score, classification_report


# Step 1: Load the data
print("reading 3 CSV's...")
g_data = pd.read_csv("g_data.csv", header=None, names=['Class', 'Fold',␣
 ↪'ProteinID', 'Sequence'])  # Protein sequences
occur_data = pd.read_csv("occur.csv")  # Occurrence features
attributes_data = pd.read_csv("attributes.csv", skiprows=1)  # Physicochemical␣
 ↪properties


print("populating Amino Acid dictionary...")
# Process the attributes data to create a dictionary for easy access
amino_acids = 'ACDEFGHIKLMNPQRSTVWY'
attributes_dict = {}

# Process each physicochemical property
for i in range(1, 10):  # Using first 9 properties for simplicity, can be␣
 ↪extended
    property_name = attributes_data.iloc[i-1, 1]
    property_values = {}
    for j, aa in enumerate(amino_acids):
        property_values[aa] = float(attributes_data.iloc[i-1, j+2])
    attributes_dict[property_name] = property_values

print(f"\nHere's a preview of what the data looks like:\n")

print("=== ATTRIBUTES DICT === \n" + str(attributes_dict) + '\n')

print("=== GDATA === \n" + str(g_data.head())+ '\n')
print("=== OCCUR_DATA === \n" + str(occur_data.head())+ '\n')
print("=== ATRRIBUTES_DATA === \n" + str(attributes_data.head())+ '\n')
```

```
reading 3 CSV's…
populating Amino Acid dictionary…

Here's a preview of what the data looks like:

=== ATTRIBUTES DICT ===
{'structure derived hydrophobicity value': {'A': 0.5, 'C': 2.3, 'D': -1.0, 'E':
-0.9, 'F': 1.3, 'G': 0.3, 'H': 0.8, 'I': 1.8, 'K': -1.2, 'L': 0.9, 'M': 0.8,
'N': -0.1, 'P': -0.6, 'Q': -0.7, 'R': -0.3, 'S': -0.4, 'T': -0.1, 'V': 1.0, 'W':
```

1.9, 'Y': 0.9}, 'Polarizability': {'A': 0.05, 'C': 0.13, 'D': 0.11, 'E': 0.15, 'F': 0.29, 'G': 0.0, 'H': 0.23, 'I': 0.19, 'K': 0.22, 'L': 0.19, 'M': 0.22, 'N': 0.13, 'P': 0.13, 'Q': 0.18, 'R': 0.29, 'S': 0.06, 'T': 0.11, 'V': 0.14, 'W': 0.41, 'Y': 0.3}, 'Normalized frequency of alpha-helix': {'A': 1.42, 'C': 0.7, 'D': 1.01, 'E': 1.51, 'F': 1.13, 'G': 0.57, 'H': 1.0, 'I': 1.08, 'K': 1.16, 'L': 1.21, 'M': 1.45, 'N': 0.67, 'P': 0.57, 'Q': 1.11, 'R': 0.98, 'S': 0.77, 'T': 0.83, 'V': 1.06, 'W': 1.08, 'Y': 0.69}, 'Normalized frequency of beta-strand': {'A': 0.83, 'C': 1.19, 'D': 0.54, 'E': 0.37, 'F': 1.38, 'G': 0.75, 'H': 0.87, 'I': 1.6, 'K': 0.74, 'L': 1.3, 'M': 1.05, 'N': 0.89, 'P': 0.55, 'Q': 1.1, 'R': 0.93, 'S': 0.75, 'T': 1.19, 'V': 1.7, 'W': 1.37, 'Y': 1.47}, 'Normalized frequency of turn': {'A': 0.66, 'C': 1.19, 'D': 1.46, 'E': 0.74, 'F': 0.6, 'G': 1.56, 'H': 0.95, 'I': 0.47, 'K': 1.01, 'L': 0.59, 'M': 0.6, 'N': 1.56, 'P': 1.52, 'Q': 0.98, 'R': 0.95, 'S': 1.43, 'T': 0.96, 'V': 0.5, 'W': 0.96, 'Y': 1.14}, 'Hydrophobicity at ph 7.5 by HPLC': {'A': 0.35, 'C': 0.76, 'D': -2.15, 'E': -1.95, 'F': 1.69, 'G': 0.0, 'H': -0.65, 'I': 1.83, 'K': -1.54, 'L': 1.8, 'M': 1.1, 'N': -0.99, 'P': 0.84, 'Q': -0.93, 'R': -1.5, 'S': -0.63, 'T': -0.27, 'V': 1.32, 'W': 1.35, 'Y': 0.39}, 'Size': {'A': 2.5, 'C': 3.0, 'D': 2.5, 'E': 5.0, 'F': 6.5, 'G': 0.5, 'H': 6.0, 'I': 5.5, 'K': 7.0, 'L': 5.5, 'M': 6.0, 'N': 5.0, 'P': 5.5, 'Q': 6.0, 'R': 7.5, 'S': 3.0, 'T': 5.0, 'V': 5.0, 'W': 7.0, 'Y': 7.0}, 'Consensus normalized hydrophobicity scale': {'A': 0.62, 'C': 0.29, 'D': -0.9, 'E': -0.74, 'F': 1.19, 'G': 0.48, 'H': -0.4, 'I': 1.38, 'K': -1.5, 'L': 1.06, 'M': 0.64, 'N': -0.78, 'P': 0.12, 'Q': -0.85, 'R': -2.53, 'S': -0.18, 'T': -0.05, 'V': 1.08, 'W': 0.81, 'Y': 0.26}, 'Hydrophobicity index base on helix in membrane': {'A': -1.6, 'C': -2.0, 'D': -2.1, 'E': -2.6, 'F': -3.7, 'G': -1.0, 'H': -3.0, 'I': -3.1, 'K': -3.7, 'L': -2.8, 'M': -3.4, 'N': -2.2, 'P': -1.8, 'Q': -2.9, 'R': -4.4, 'S': -1.6, 'T': -2.2, 'V': -2.6, 'W': -4.9, 'Y': -3.7}}

=== GDATA ===

	Class	Fold	ProteinID	Sequence
0	1	Fold1	>P23453	MSGEVLSQNEIDALLSAISTGEMDAEELKKEEKEKKVKVYDFKRAL…
1	1	Fold1	>P07373	MTTKKTSPDLLLVIITLLLLTIGLIMVYSASAVWADYKFDDSFFFA…
2	1	Fold1	>P12921	MIIWINGAFGSGKTQTAFELHRRLNPSYVYDPQKMGFALRSMVPQE…
3	1	Fold1	>P19579	MRRKLTFQEKLLIFIKKTKKKNPRYVAIVLPLIAVILIAATWVQRT…
4	1	Fold1	>A2RMA8	MQNLNKTEKTFFGQPRGLLTLFQTEFWERFSYYGMRAILVYYLYAL…

=== OCCUR_DATA ===

	Fold	V1	V2	V3	V4	V5	V6	V7	V8	V9	…	V11	V12	V13	V14	V15	\
0	Fold1	14	2	18	34	11	16	8	31	19	…	13	15	15	14	14	
1	Fold1	23	3	9	6	31	39	1	38	11	…	16	6	14	10	14	
2	Fold1	15	1	11	11	9	10	6	17	9	…	6	8	9	8	18	
3	Fold1	31	0	27	24	13	23	14	21	49	…	9	24	14	17	15	
4	Fold1	41	0	11	12	36	43	6	54	17	…	18	20	21	14	18	

	V16	V17	V18	V19	V20
0	24	20	27	2	6
1	27	18	31	7	11
2	12	10	8	3	6
3	16	25	40	4	14

```
4   33   29   34   10   16

[5 rows x 21 columns]

=== ATRRIBUTES_DATA ===
   Number                                Attributes     A     C     D     E  \
0       1  structure derived hydrophobicity value  0.50  2.30 -1.00 -0.90
1       2                           Polarizability  0.05  0.13  0.11  0.15
2       3     Normalized frequency of alpha-helix  1.42  0.70  1.01  1.51
3       4     Normalized frequency of beta-strand  0.83  1.19  0.54  0.37
4       5            Normalized frequency of turn  0.66  1.19  1.46  0.74

      F     G     H     I  …     M     N     P     Q     R     S     T  \
0  1.30  0.30  0.80  1.80  …  0.80 -0.10 -0.60 -0.70 -0.30 -0.40 -0.10
1  0.29  0.00  0.23  0.19  …  0.22  0.13  0.13  0.18  0.29  0.06  0.11
2  1.13  0.57  1.00  1.08  …  1.45  0.67  0.57  1.11  0.98  0.77  0.83
3  1.38  0.75  0.87  1.60  …  1.05  0.89  0.55  1.10  0.93  0.75  1.19
4  0.60  1.56  0.95  0.47  …  0.60  1.56  1.52  0.98  0.95  1.43  0.96

      V     W     Y
0  1.00  1.90  0.90
1  0.14  0.41  0.30
2  1.06  1.08  0.69
3  1.70  1.37  1.47
4  0.50  0.96  1.14

[5 rows x 22 columns]
```

2.1 Analysis

This step is pretty simple; we just read the CSV's into panda dataframes (which is essentially a matrix with labeled axes)

We also importantly create a Dictionary to store the data from the Attributes CSV, which contains different properties about the molecules. We pay a bit of computation in this step (a really tiny amount iterating through matrix) for easier access later in the program

3 Step 2: Features

[35]:
```python
def amino_acid_composition(sequence):
    """Calculate amino acid composition (frequencies) in the sequence."""
    #sequence = extract_sequence(sequence)
    amino_acids = 'ACDEFGHIKLMNPQRSTVWY'
    counts = Counter(sequence)
    composition = {aa: counts.get(aa, 0) / len(sequence) for aa in amino_acids}
    return composition
```

```
def dipeptide_composition(sequence):
    """Calculate dipeptide (2-mer) composition in the sequence."""
    #sequence = extract_sequence(sequence)
    amino_acids = 'ACDEFGHIKLMNPQRSTVWY'
    dipeptides = [aa1+aa2 for aa1 in amino_acids for aa2 in amino_acids]

    # Count dipeptides
    dip_counts = {}
    for i in range(len(sequence)-1):
        dipeptide = sequence[i:i+2]
        if all(aa in amino_acids for aa in dipeptide):
            dip_counts[dipeptide] = dip_counts.get(dipeptide, 0) + 1

    # Normalize by total number of dipeptides
    total_dipeptides = max(1, len(sequence)-1)  # Avoid division by zero
    dip_composition = {dip: dip_counts.get(dip, 0) / total_dipeptides for dip␣
 ↪in dipeptides}
    return dip_composition

def avg_physicochemical_properties(sequence):
    """Calculate average physicochemical properties for the sequence."""
    #sequence = extract_sequence(sequence)
    properties = {}

    # Calculate average value for each property
    for prop_name, prop_values in attributes_dict.items():
        avg_value = 0
        count = 0
        for aa in sequence:
            if aa in prop_values:
                avg_value += prop_values[aa]
                count += 1
        if count > 0:
            avg_value /= count
        properties[f"avg_{prop_name}"] = avg_value

    return properties

# After we define our functione, we apply feature extraction to all sequences␣
 ↪by accessing the Sequence Column
g_data['aa_features'] = g_data['Sequence'].apply(amino_acid_composition)
g_data['dip_features'] = g_data['Sequence'].apply(dipeptide_composition)
g_data['phys_features'] = g_data['Sequence'].
 ↪apply(avg_physicochemical_properties)

print("After feature extraction, g_data will have 3 new columns for Amino Acid␣
 ↪Composition, Physicochemical Properties, and Dipeptide Composition")
```

```
print(f"Heres a sample of the matrix now:\n\n{g_data.head()}")

# Convert extracted features into DataFrames
aa_features = pd.DataFrame(g_data['aa_features'].tolist())
dip_features = pd.DataFrame(g_data['dip_features'].tolist())
phys_features = pd.DataFrame(g_data['phys_features'].tolist())


# Step 3: Combine features
# Merge occurrence features and all sequence-derived features
combined_data = pd.concat([
    occur_data.iloc[:, 1:],  # Occurrence features
    aa_features,             # Amino acid composition
    phys_features,           # Physicochemical properties
    dip_features             # Dipeptide composition
], axis=1)

print(f"\nNext we extract these features, take them out of g_data, and put them␣
 ↪into their own matrix\n")

print(f"Recall that there are 523 proteins and 449 different feature␣
 ↪combinations, thus the matrix is size {combined_data.shape}")


# Add labels and encode them numerically
# Extract the fold information
label_encoder = LabelEncoder()
labels = label_encoder.fit_transform(g_data['Fold'])  # Now this will get␣
 ↪Fold1, Fold2, etc.
fold_mapping = dict(zip(range(len(label_encoder.classes_)), label_encoder.
 ↪classes_))
print(f"\nRecall that there are 4 folds in g_data\nLabel mapping:␣
 ↪{fold_mapping}\n")


# Step 4: Normalize and select features
scaler = StandardScaler()
normalized_features = scaler.fit_transform(combined_data)


# Note: These train/test split operations have been moved after feature␣
 ↪selection
# Feature selection using Random Forest
print("Performing feature selection...")
selector = SelectFromModel(
    RandomForestClassifier(n_estimators=100, random_state=42),
```

```
    threshold="median"
)
selector.fit(normalized_features, labels)
selected_features = selector.transform(normalized_features)
print(f"Selected {selected_features.shape[1]} out of {normalized_features.
 ↪shape[1]} features")

print(f"We use Random Forest Classifier to identify these features, note that␣
 ↪the matrix is now {selected_features.shape}")
```

After feature extraction, g_data will have 3 new columns for Amino Acid Composition, Physicochemical Properties, and Dipeptide Composition
Heres a sample of the matrix now:

```
   Class   Fold ProteinID                                              Sequence  \
0      1  Fold1   >P23453  MSGEVLSQNEIDALLSAISTGEMDAEELKKEEKEKKVKVYDFKRAL…
1      1  Fold1   >P07373  MTTKKTSPDLLLVIITLLLLTIGLIMVYSASAVWADYKFDDSFFFA…
2      1  Fold1   >P12921  MIIWINGAFGSGKTQTAFELHRRLNPSYVYDPQKMGFALRSMVPQE…
3      1  Fold1   >P19579  MRRKLTFQEKLLIFIKKTKKKNPRYVAIVLPLIAVILIAATWVQRT…
4      1  Fold1   >A2RMA8  MQNLNKTEKTFFGQPRGLLTLFQTEFWERFSYYGMRAILVYYLYAL…

                                        aa_features  \
0  {'A': 0.04216867469879518, 'C': 0.006024096385…
1  {'A': 0.06284153005464481, 'C': 0.008196721311…
2  {'A': 0.07614213197969544, 'C': 0.005076142131…
3  {'A': 0.07542579075425791, 'C': 0.0, 'D': 0.06…
4  {'A': 0.08249496981891348, 'C': 0.0, 'D': 0.02…

                                       dip_features  \
0  {'AA': 0.0, 'AC': 0.0, 'AD': 0.003021148036253…
1  {'AA': 0.0, 'AC': 0.0, 'AD': 0.002739726027397…
2  {'AA': 0.01020408163265306, 'AC': 0.0, 'AD': 0…
3  {'AA': 0.0024390243902439024, 'AC': 0.0, 'AD':…
4  {'AA': 0.004032258064516129, 'AC': 0.0, 'AD': …

                                      phys_features
0  {'avg_structure derived hydrophobicity value':…
1  {'avg_structure derived hydrophobicity value':…
2  {'avg_structure derived hydrophobicity value':…
3  {'avg_structure derived hydrophobicity value':…
4  {'avg_structure derived hydrophobicity value':…
```

Next we extract these features (take them out of g_data) and put them into their own matrix

Recall that there are 523 proteins and 449 different feature combinations, thus the matrix is size (523, 449)

```
Recall that there are 4 folds in g_data
Label mapping: {0: 'Fold1', 1: 'Fold2', 2: 'Fold3', 3: 'Fold4'}

Performing feature selection…
Selected 225 out of 449 features
We use Random Forest Classifier to identify these features, note that the matrix
is now (523, 225)
```

3.1 Analysis

This step is a jump in complexity but it is ultimately just basic feature extraction. To explain the process, our CSV g_data starts sa list of 523 protein sequences and their respective folds. We read these sequences and extrapolate certain information about them; in our case it is simply their Amino Acid struture (Single Occurances), DiPeptide Structure (Pair Occurances), and Physiochemical Properties (Attributes)

- Example: For a sequence "ACDKLLM", the amino acid composition would be:
- {'A': 0.143, 'C': 0.143, 'D': 0.143, 'K': 0.143, 'L': 0.286, 'M': 0.143, …} (with zeros for all other amino acids not present)
- Example: For a sequence "ACDKLLM", the dipeptides are "AC", "CD", "DK", "KL", "LL", "LM"
- Each dipeptide frequency would be $1/6 = 0.167$ (with zeros for all other combinations)

3.1.1 Physicochemical Properties

Using the attributes.csv file, we calculated the average value of 9 different physicochemical properties for each protein sequence. These properties include hydrophobicity, polarity, charge, and other biochemical characteristics that influence protein folding and function.

3.1.2 Feature Combination and Selection

The next critical step was to combine all extracted features into a unified feature matrix:

```
Recall that there are 523 proteins and 449 different feature combinations, thus the matrix is s
```

The high-dimensional matrix contains: - 20 amino acid composition features - 400 dipeptide composition features - 9 averaged physicochemical properties - 20 occurrence features from the original `occur.csv` file

To improve model performance and reduce overfitting, we applied feature selection using a Random Forest classifier, which identified the most informative features:

```
Selected 225 out of 449 features
```

This dimensionality reduction cut our feature space nearly in half while preserving the most predictive attributes, significantly improving both model performance and training efficiency.

The resulting feature matrix (523 rows $\times$ 225 columns) provides a comprehensive numerical representation of the protein sequences, capturing compositional, sequential, and physicochemical information relevant to cellular localization prediction.

4 Step 2.5: Split Data into Training/Testing

[38]:
```
# Step 5: Split the data into training and testing sets (with stratification)
X_train_main, X_test_independent, y_train_main, y_test_independent =␣
 ↪train_test_split(
    selected_features,        # Use selected features instead of␣
 ↪normalized_features
    labels,
    test_size=0.2,          # 20% for independent testing
    random_state=42,
    stratify=labels         # ensure balanced class distribution
)

# Second split: create validation set from training data
X_train, X_test, y_train, y_test = train_test_split(
    X_train_main,
    y_train_main,
    test_size=0.25,
    random_state=42,
    stratify=y_train_main
)
```

In this step, we did two things: - First we created an independent test set that would remain completely untouched during model development ***20% of data*** - Second we created validation and training sets *__80% of remaining data__ - A training set (75% of X_train_main, or approximately 314 samples) for model fitting - A validation set (25% of X_train_main, or approximately 104 samples) for hyperparameter tuning

A suprisingly important function is that both splitting operations used stratification (stratify=labels and stratify=y_train_main), which makes sure that **the class distribution in the original dataset is preserved in all subsets (Folds).**

This is particularly important because our dataset has four protein location classes (Folds) that may be imbalanced. If the original dataset contained: - 40% Fold1 - 30% Fold2 - 20% Fold3 - 10% Fold4

Then each subset (training, validation, and independent test) would **maintain these same proportions, preventing sampling bias** and ensuring that models are evaluated on properly representative data.

5 Step 3: Models; Hyperparameter Tuning and Training

[40]:
```
"""
-------------------------------------K-Nearest Neighbors (KNN) with␣
 ↪hyperparameter tuning-------------------------------------
"""
print("\nTuning KNN hyperparameters...")
knn_params = {
```

```
    'n_neighbors': [3, 5, 7, 9],
    'weights': ['uniform', 'distance'],
    'p': [1, 2]  # p=1 is Manhattan distance, p=2 is Euclidean
}
knn = GridSearchCV(
    KNeighborsClassifier(),
    knn_params,
    cv=StratifiedKFold(n_splits=5, shuffle=True, random_state=42),
    scoring='accuracy'
)
knn.fit(X_train, y_train)
print(f"Best KNN parameters: {knn.best_params_}")
y_pred_knn = knn.predict(X_test)
print("KNN Accuracy:", round(accuracy_score(y_test, y_pred_knn), 2))
print(classification_report(y_test, y_pred_knn, zero_division=0))

"""
-------------------------------------Support Vector Machine (SVM) with␣
 ↪hyperparameter tuning-------------------------------------
"""
print("\nTuning SVM hyperparameters...")
svm_params = {
    'C': [10],
    'kernel': ['rbf'],
    'gamma': ['scale']
}
svm = GridSearchCV(
    SVC(probability=True),
    svm_params,
    cv=StratifiedKFold(n_splits=5, shuffle=True, random_state=42),
    scoring='accuracy'
)
svm.fit(X_train, y_train)
print(f"Best SVM parameters: {svm.best_params_}")
y_pred_svm = svm.predict(X_test)
print("SVM Accuracy:", round(accuracy_score(y_test, y_pred_svm), 2))
print(classification_report(y_test, y_pred_svm, zero_division=0))

"""
-------------------------------------Random Forest with hyperparameter␣
 ↪tuning-------------------------------------
"""
print("\nTuning Random Forest hyperparameters...")
rf_params = {
    'n_estimators': [100],
    'max_depth': [None],
    'min_samples_split': [2],
```

```
    'min_samples_leaf': [1]
}
rf = GridSearchCV(
    RandomForestClassifier(random_state=42),
    rf_params,
    cv=StratifiedKFold(n_splits=5, shuffle=True, random_state=42),
    scoring='accuracy'
)
rf.fit(X_train, y_train)
print(f"Best Random Forest parameters: {rf.best_params_}")
y_pred_rf = rf.predict(X_test)
print("Random Forest Accuracy:", round(accuracy_score(y_test, y_pred_rf), 2))
print(classification_report(y_test, y_pred_rf, zero_division=0))

"""
Naïve Bayes with parameter exploration
"""
print("\nTraining Naive Bayes model...")
nb = GaussianNB()
nb.fit(X_train, y_train)
y_pred_nb = nb.predict(X_test)
print("Naïve Bayes Accuracy:", round(accuracy_score(y_test, y_pred_nb), 2))
print(classification_report(y_test, y_pred_nb, zero_division=0))

"""
-------------------------------------Artificial Neural Network (ANN) with␣
 ↪hyperparameter tuning-----------------------------------------
"""
print("\nTuning Neural Network hyperparameters...")
ann_params = {
    'hidden_layer_sizes': [(100, 50)],
    'activation': ['relu'],
    'alpha': [0.0001],
    'learning_rate': ['adaptive'],
    'max_iter': [1000]
}
ann = GridSearchCV(
    MLPClassifier(random_state=42),
    ann_params,
    cv=StratifiedKFold(n_splits=3, shuffle=True, random_state=42),  # Using␣
 ↪3-fold to save time
    scoring='accuracy'
)
ann.fit(X_train, y_train)
print(f"Best Neural Network parameters: {ann.best_params_}")
y_pred_ann = ann.predict(X_test)
print("ANN Accuracy:", round(accuracy_score(y_test, y_pred_ann), 2))
```

```
print(classification_report(y_test, y_pred_ann, zero_division=0))


"""
-------------------------------------Bagging␣
 ↪Classifier-------------------------------------
"""
bagging = BaggingClassifier(estimator = KNeighborsClassifier(), n_estimators =␣
 ↪50, random_state = 42)
bagging.fit(X_train, y_train)
y_pred_bagging = bagging.predict(X_test)
print("Bagging Accuracy:", round(accuracy_score(y_test, y_pred_bagging), 2))
print(classification_report(y_test, y_pred_bagging, zero_division=0))
# AdaBoost Classifier
print("\nTraining AdaBoost Classifier...")
ada = AdaBoostClassifier(
    estimator=RandomForestClassifier(max_depth=3, random_state=42),
    n_estimators=100,
    learning_rate=0.1,
    random_state=42
)
ada.fit(X_train, y_train)
y_pred_ada = ada.predict(X_test)
print("AdaBoost Accuracy:", round(accuracy_score(y_test, y_pred_ada), 2))
print(classification_report(y_test, y_pred_ada, zero_division=0))


"""
-------------------------------------Stacking␣
 ↪Classifier-------------------------------------
"""
print("\nTraining Stacking Classifier...")
estimators = [
    ('knn', KNeighborsClassifier(n_neighbors=5)),
    ('rf', RandomForestClassifier(n_estimators=100, random_state=42)),
    ('svm', SVC(probability=True, kernel='rbf', random_state=42))
]
stacking = StackingClassifier(
    estimators=estimators,
    final_estimator=RandomForestClassifier(n_estimators=100, random_state=42),
    cv=5,
    stack_method='predict_proba'
)
stacking.fit(X_train, y_train)
y_pred_stacking = stacking.predict(X_test)
print("Stacking Classifier Accuracy:", round(accuracy_score(y_test,␣
 ↪y_pred_stacking), 2))
```

```
print(classification_report(y_test, y_pred_stacking, zero_division=0))


"""
-------------------------------------Voting␣
 ↪Classifier-------------------------------------
"""
print("\nTraining Voting Classifier...")
voting = VotingClassifier(
    estimators=[
        ('knn', knn.best_estimator_),
        ('rf', rf.best_estimator_),
        ('svm', svm.best_estimator_),
        ('nb', GaussianNB()),
        ('ann', ann.best_estimator_),
        ('ada', ada)
    ],
    voting='soft'
)
voting.fit(X_train, y_train)
y_pred_voting = voting.predict(X_test)
print("Voting Classifier Accuracy:", round(accuracy_score(y_test,␣
 ↪y_pred_voting), 2))
print(classification_report(y_test, y_pred_voting, zero_division=0))
```

```
Tuning KNN hyperparameters…
Best KNN parameters: {'n_neighbors': 9, 'p': 2, 'weights': 'uniform'}
KNN Accuracy: 0.7
              precision    recall  f1-score   support

           0       0.75      0.60      0.67        35
           1       1.00      0.25      0.40         4
           2       0.67      0.93      0.78        42
           3       0.67      0.50      0.57        24

    accuracy                           0.70       105
   macro avg       0.77      0.57      0.60       105
weighted avg       0.71      0.70      0.68       105


Tuning SVM hyperparameters…
Best SVM parameters: {'C': 10, 'gamma': 'scale', 'kernel': 'rbf'}
SVM Accuracy: 0.75
              precision    recall  f1-score   support

           0       0.73      0.69      0.71        35
```

```
           1       0.00      0.00      0.00         4
           2       0.80      0.93      0.86        42
           3       0.70      0.67      0.68        24

    accuracy                           0.75       105
   macro avg       0.55      0.57      0.56       105
weighted avg       0.72      0.75      0.73       105


Tuning Random Forest hyperparameters…
Best Random Forest parameters: {'max_depth': None, 'min_samples_leaf': 1,
'min_samples_split': 2, 'n_estimators': 100}
Random Forest Accuracy: 0.66
              precision    recall  f1-score   support

           0       0.70      0.54      0.61        35
           1       0.00      0.00      0.00         4
           2       0.64      0.86      0.73        42
           3       0.64      0.58      0.61        24

    accuracy                           0.66       105
   macro avg       0.50      0.50      0.49       105
weighted avg       0.64      0.66      0.64       105


Training Naive Bayes model…
Naïve Bayes Accuracy: 0.68
              precision    recall  f1-score   support

           0       0.72      0.51      0.60        35
           1       1.00      0.25      0.40         4
           2       0.70      0.90      0.79        42
           3       0.56      0.58      0.57        24

    accuracy                           0.68       105
   macro avg       0.75      0.56      0.59       105
weighted avg       0.69      0.68      0.66       105


Tuning Neural Network hyperparameters…
Best Neural Network parameters: {'activation': 'relu', 'alpha': 0.0001,
'hidden_layer_sizes': (100, 50), 'learning_rate': 'adaptive', 'max_iter': 1000}
ANN Accuracy: 0.7
              precision    recall  f1-score   support

           0       0.72      0.66      0.69        35
           1       0.00      0.00      0.00         4
           2       0.74      0.81      0.77        42
```

```
           3       0.59      0.67      0.63        24

    accuracy                           0.70       105
   macro avg       0.51      0.53      0.52       105
weighted avg       0.67      0.70      0.68       105

Bagging Accuracy: 0.66
              precision    recall  f1-score   support

           0       0.74      0.57      0.65        35
           1       1.00      0.25      0.40         4
           2       0.64      0.88      0.74        42
           3       0.58      0.46      0.51        24

    accuracy                           0.66       105
   macro avg       0.74      0.54      0.57       105
weighted avg       0.67      0.66      0.64       105


Training AdaBoost Classifier…
AdaBoost Accuracy: 0.68
              precision    recall  f1-score   support

           0       0.64      0.66      0.65        35
           1       0.00      0.00      0.00         4
           2       0.74      0.88      0.80        42
           3       0.58      0.46      0.51        24

    accuracy                           0.68       105
   macro avg       0.49      0.50      0.49       105
weighted avg       0.64      0.68      0.65       105


Training Stacking Classifier…
Stacking Classifier Accuracy: 0.74
              precision    recall  f1-score   support

           0       0.77      0.66      0.71        35
           1       0.00      0.00      0.00         4
           2       0.80      0.86      0.83        42
           3       0.63      0.79      0.70        24

    accuracy                           0.74       105
   macro avg       0.55      0.58      0.56       105
weighted avg       0.72      0.74      0.73       105


Training Voting Classifier…
```

```
Voting Classifier Accuracy: 0.7
              precision    recall  f1-score   support

           0       0.71      0.57      0.63        35
           1       0.00      0.00      0.00         4
           2       0.71      0.88      0.79        42
           3       0.64      0.67      0.65        24

    accuracy                           0.70       105
   macro avg       0.52      0.53      0.52       105
weighted avg       0.67      0.70      0.68       105
```

5.1 Analysis

This step is obviously crucial, however, I think the code is self-explanatory. A key point in the project for this step was tuning the models and model specific parameters to boost the accuracy. We did see some improvements, with SVM reaching 75% accuracy, a relatively big improvement over the previous 65%. As you can see the ideal params are

```
{'C': 10, 'gamma': 'scale', 'kernel': 'rbf'}
```

These parameters indicate that a radial basis function kernel with moderately high regularization strength performed best for protein localization prediction, effectively handling the non-linear relationships in our feature space. This being said, we did not utilize much methodology to achieve this improvement. But simply tweaking parameters to test them is always a great idea and with more time and rigor this could likely be even higher.

5.1.1 Models Used

- KNN For KNN, we identified optimal parameters of 9 neighbors with uniform weighting and Euclidean distance (p=2), achieving 70% accuracy. This suggests that protein localization patterns are best captured by examining several nearby reference proteins.
- SVM
- Random Forest
- Naive Bayes
- ANN Our ANN architecture with hidden layer sizes of (100, 50), ReLU activation, and adaptive learning rate yielded 70% accuracy, showing strong performance in capturing complex feature relationships.

5.1.2 Here are our ensemble Methods

- Bagging (66% accuracy): Used multiple KNN classifiers on bootstrapped samples
- AdaBoost (68% accuracy): Sequentially focused on misclassified proteins
- Stacking Our second-best model at 74% accuracy was a stacking ensemble that combined KNN, Random Forest, and SVM with a Random Forest meta-learner. This approach leveraged the strengths of multiple algorithms to improve overall prediction reliability.
- Voting Classifier (70% accuracy): Combined predictions from our six best models

6 Step 4: Results, Interpretation, Cross-Validation

To ensure robust performance estimation, we employed **both stratified k-fold cross-validation on the training data and evaluation on a completely independent test set.**

- Random Forest Cross-Validation Balanced Accuracy: 0.52
- SVM Cross-Validation Balanced Accuracy: 0.56
- Voting Cross-Validation Balanced Accuracy: 0.56

The balanced accuracy scores account for potential class imbalance, providing a more realistic performance measure. These cross-validation results highlight that while SVM and Voting classifiers showed strong performance, they still faced challenges with underrepresented classes.

6.0.1 When evaluated on the completely independent test set, our models showed consistency in their performance:

- K-Nearest Neighbors: 0.67
- Support Vector Machine: 0.69
- Random Forest: 0.64
- Naive Bayes: 0.64
- Artificial Neural Network: 0.70
- Bagging: 0.66

The ANN performed best on independent data, demonstrating strong generalization capability, closely followed by SVM.

[44]:
```
print("\nPerforming cross-validation with stratified k-fold...")
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

print("We only need to cross-validate our best models to save time and energy...
 ↪")

cv_scores_rf = cross_val_score(rf, selected_features, labels, cv=cv,␣
 ↪scoring='balanced_accuracy')
print("Random Forest Cross-Validation Balanced Accuracy:", round(cv_scores_rf.
 ↪mean(), 2))

cv_scores_svm = cross_val_score(svm, selected_features, labels, cv=cv,␣
 ↪scoring='balanced_accuracy')
print("SVM Cross-Validation Balanced Accuracy:", round(cv_scores_svm.mean(), 2))

cv_scores_voting = cross_val_score(voting, selected_features, labels, cv=cv,␣
 ↪scoring='balanced_accuracy')
print("Voting Cross-Validation Balanced Accuracy:", round(cv_scores_voting.
 ↪mean(), 2))


# Evaluate all models on the independent test set
def evaluate_model(model, model_name):
```

```
    y_pred = model.predict(X_test_independent)
    accuracy = accuracy_score(y_test_independent, y_pred)
    report = classification_report(y_test_independent, y_pred, zero_division=0)

    print(f"\n{model_name} Performance on Independent Test Set:")
    print(f"Accuracy: {accuracy:.2f}")
    print("Detailed Classification Report:")
    print(report)

    return accuracy

# Evaluate each model
independent_results = {
    'KNN': evaluate_model(knn, "K-Nearest Neighbors"),
    'SVM': evaluate_model(svm, "Support Vector Machine"),
    'RF': evaluate_model(rf, "Random Forest"),
    'NB': evaluate_model(nb, "Naive Bayes"),
    'ANN': evaluate_model(ann, "Artificial Neural Network"),
    'Bagging': evaluate_model(bagging, "Bagging")
}

# Compare model performances
print("\nModel Performance Comparison on Independent Test Set:")
for model_name, accuracy in independent_results.items():
    print(f"{model_name}: {accuracy:.2f}")


# Save processed data
processed_data = pd.concat([pd.DataFrame(normalized_features), pd.
  ↪Series(labels, name='Label')], axis=1)
processed_data.to_csv("Data/processed_data.csv", index=False)


# Print summary of best models (based on test accuracy)
print("\n=== Model Accuracy Summary ===")
model_accuracies = {
    "KNN": round(accuracy_score(y_test, y_pred_knn), 2),
    "SVM": round(accuracy_score(y_test, y_pred_svm), 2),
    "Random Forest": round(accuracy_score(y_test, y_pred_rf), 2),
    "Naive Bayes": round(accuracy_score(y_test, y_pred_nb), 2),
    "ANN": round(accuracy_score(y_test, y_pred_ann), 2),
    "Bagging": round(accuracy_score(y_test, y_pred_bagging), 2),
    "AdaBoost": round(accuracy_score(y_test, y_pred_ada), 2),
    "Stacking": round(accuracy_score(y_test, y_pred_stacking), 2),
    "Voting": round(accuracy_score(y_test, y_pred_voting), 2)
}
```

```python
# Sort models by accuracy
sorted_models = sorted(model_accuracies.items(), key=lambda x: x[1],␣
 ↪reverse=True)


# Print models in order of performance
for model_name, accuracy in sorted_models:
    print(f"{model_name} Accuracy: {accuracy}")

print("\n=== Accuracy Improvement Summary ===")
print("Original KNN Accuracy: 0.70 → New KNN Accuracy: " +␣
 ↪str(model_accuracies["KNN"]))
print("Original SVM Accuracy: 0.67 → New SVM Accuracy: " +␣
 ↪str(model_accuracies["SVM"]))
print("Original Random Forest Accuracy: 0.71 → New Random Forest Accuracy: " +␣
 ↪str(model_accuracies["Random Forest"]))
print("Original Naive Bayes Accuracy: 0.64 → New Naive Bayes Accuracy: " +␣
 ↪str(model_accuracies["Naive Bayes"]))
print("Original ANN Accuracy: 0.68 → New ANN Accuracy: " +␣
 ↪str(model_accuracies["ANN"]))
print("Original Bagging Accuracy: 0.68 → New Bagging Accuracy: " +␣
 ↪str(model_accuracies["Bagging"]))

print("\nBest Model: " + sorted_models[0][0] + " with accuracy " +␣
 ↪str(sorted_models[0][1]))
print("Average accuracy improvement: " + str(round(((model_accuracies["KNN"] +␣
 ↪model_accuracies["SVM"] + model_accuracies["Random Forest"] +␣
 ↪model_accuracies["Naive Bayes"] + model_accuracies["ANN"] +␣
 ↪model_accuracies["Bagging"])/6 - 0.68), 2)))
```

```
Performing cross-validation with stratified k-fold…
We only need to cross-validate our best models to save time and energy…
Random Forest Cross-Validation Balanced Accuracy: 0.52
SVM Cross-Validation Balanced Accuracy: 0.56
Voting Cross-Validation Balanced Accuracy: 0.56

K-Nearest Neighbors Performance on Independent Test Set:
Accuracy: 0.67
Detailed Classification Report:
              precision    recall  f1-score   support

           0       0.76      0.54      0.63        35
           1       0.00      0.00      0.00         3
           2       0.63      0.93      0.75        42
           3       0.71      0.48      0.57        25
```

```
    accuracy                           0.67       105
   macro avg       0.52      0.49      0.49       105
weighted avg       0.67      0.67      0.65       105


Support Vector Machine Performance on Independent Test Set:
Accuracy: 0.69
Detailed Classification Report:
              precision    recall  f1-score   support

           0       0.70      0.54      0.61        35
           1       0.00      0.00      0.00         3
           2       0.71      0.86      0.77        42
           3       0.63      0.68      0.65        25

    accuracy                           0.69       105
   macro avg       0.51      0.52      0.51       105
weighted avg       0.67      0.69      0.67       105


Random Forest Performance on Independent Test Set:
Accuracy: 0.64
Detailed Classification Report:
              precision    recall  f1-score   support

           0       0.77      0.49      0.60        35
           1       0.00      0.00      0.00         3
           2       0.61      0.79      0.69        42
           3       0.59      0.68      0.63        25

    accuracy                           0.64       105
   macro avg       0.49      0.49      0.48       105
weighted avg       0.64      0.64      0.62       105


Naive Bayes Performance on Independent Test Set:
Accuracy: 0.64
Detailed Classification Report:
              precision    recall  f1-score   support

           0       0.95      0.51      0.67        35
           1       0.33      0.33      0.33         3
           2       0.60      0.74      0.66        42
           3       0.55      0.68      0.61        25

    accuracy                           0.64       105
   macro avg       0.61      0.57      0.57       105
```

```
weighted avg       0.69      0.64      0.64       105


Artificial Neural Network Performance on Independent Test Set:
Accuracy: 0.70
Detailed Classification Report:
              precision    recall  f1-score   support

           0       0.66      0.60      0.63        35
           1       0.00      0.00      0.00         3
           2       0.77      0.81      0.79        42
           3       0.64      0.72      0.68        25

    accuracy                           0.70       105
   macro avg       0.52      0.53      0.52       105
weighted avg       0.68      0.70      0.69       105


Bagging Performance on Independent Test Set:
Accuracy: 0.66
Detailed Classification Report:
              precision    recall  f1-score   support

           0       0.72      0.51      0.60        35
           1       0.00      0.00      0.00         3
           2       0.62      0.93      0.74        42
           3       0.71      0.48      0.57        25

    accuracy                           0.66       105
   macro avg       0.51      0.48      0.48       105
weighted avg       0.66      0.66      0.63       105


Model Performance Comparison on Independent Test Set:
KNN: 0.67
SVM: 0.69
RF: 0.64
NB: 0.64
ANN: 0.70
Bagging: 0.66

=== Model Accuracy Summary ===
SVM Accuracy: 0.75
Stacking Accuracy: 0.74
KNN Accuracy: 0.7
ANN Accuracy: 0.7
Voting Accuracy: 0.7
Naive Bayes Accuracy: 0.68
```

```
AdaBoost Accuracy: 0.68
Random Forest Accuracy: 0.66
Bagging Accuracy: 0.66

=== Accuracy Improvement Summary ===
Original KNN Accuracy: 0.70 → New KNN Accuracy: 0.7
Original SVM Accuracy: 0.67 → New SVM Accuracy: 0.75
Original Random Forest Accuracy: 0.71 → New Random Forest Accuracy: 0.66
Original Naive Bayes Accuracy: 0.64 → New Naive Bayes Accuracy: 0.68
Original ANN Accuracy: 0.68 → New ANN Accuracy: 0.7
Original Bagging Accuracy: 0.68 → New Bagging Accuracy: 0.66

Best Model: SVM with accuracy 0.75
Average accuracy improvement: 0.01
```

7 Final Thoughts and Conclusions

The detailed classification reports revealed important patterns across models:

1. Class Imbalance Issues: All models struggled with class 1 (likely the minority class), often showing zero precision and recall. This indicates insufficient training examples for this location.
2. Strong Performance for Class 2: Most models exhibited high recall (up to 93%) for class 2, suggesting this subcellular location has distinctive features.
3. Precision-Recall Trade-offs: Models generally showed imbalances between precision and recall. For example, SVM on the validation set achieved high precision for class 2 (0.80) and class 0 (0.73), but lower for class 3 (0.70).
4. Weighted vs. Macro Averages: The difference between weighted and macro average F1-scores (SVM: weighted=0.73 vs macro=0.56) confirms the impact of class imbalance on overall metrics.

Our systematic approach resulted in meaningful improvements:

```
SVM improved from 67% to 75% accuracy
Naive Bayes improved from 64% to 68% accuracy
ANN improved from 68% to 70% accuracy
```

While Random Forest and Bagging showed slight decreases in performance, the average improvement across all models was +0.01, demonstrating the value of our feature engineering and hyperparameter optimization efforts.

8 Takeaways

This project ultimately shows both the power and limitations of machine learning in bioinformatics. If we were to takeaway some main points to apply to future work in both ML and Biological Data.

1. Choosing Features is hugely important. Although we did not experiment with the actual effects of this, a considerable amount of the process is simply cleaning data and identifying actionable features. Organizing the data by hand like this puts less work on the model, so

it can work with a cleaner data pool, be more efficient, and hopefully recognize meaningful patterns easier (essentially we removed columns with the most 0s, some decisions were more complex than this but we used scikit learn to handle this).

2. Class Imbalance Challenges: The consistent difficulty in classifying proteins from the minority class highlights a fundamental challenge in biological datasets. Future work should focus on specialized techniques for imbalanced data, such as oversampling or class-weighted loss functions.
3. Model Selection Trade-offs: While SVM performed best on validation data (75%), ANN showed superior generalization on the independent test set (70%), suggesting that model selection should consider both validation performance and generalization capability.
4. Ensemble Benefits: Stacking and voting classifiers demonstrated strong performance, reenforcing the value of combining multiple modeling approaches when tackling complex (biological) classification problems. Although you should generally start simple with new problems, its evident in this case that a combination does achieve great results, and it most cases will.