Speech Task № 6

Vladyslav Bondarenko, University of Sheffield May 23, 2020

Initial Implementation

Main part of initial implementation for KM is shown in Listing 1 where **X** are input features and **centers** are initialized to unique random rows in **X**. Running the function demonstrated comparable results with **scikit-learn** implementation of the algorithm for most of the seeds. Nevertheless, trying on many different seeds demonstrates that personal implementation relies on a lucky random initialization and in some rare cases results in poor clusters such as demonstrated in Fig. 1. It appears like **a** is being mistaken for **u** and there are only 3 clusters generated in the plot. A small amount of this error could also be attributed to the way clusters are matched to labels but it still clearly demonstrates poor performance. Inspecting scikitlearn documentation shows that their implementation uses a sophisticated initialization algorithm by default; further confirming that poor results can be attributed to fully random initialisation. Thus, implementing a more robust initialization could solve the issue.

Running the code by hand for each different implementation for many different seeds and visually comparing the confusion matrices would become tedious. It would be nice to summarize the results from confusion matrix in a single number. Accuracy could be used for that but would not generalize well for problems where classes are imbalanced. Therefore, **F1** score is used for evaluation, representing the weighted average of precision and recall. Evaluation is run for over 100 iterations and mean and variance of F1 scores are reported. Score for initial implementation confirms poor performance with mean F1 = 0.78 and variance = 0.02. In comparison **scikit-learn** implementation achieves 0.98 and 0.0 for mean and variance respectively demonstrating very stable performance.

Listing 1: Main loop of K-means clustering

```
rand_idx = random.sample(range(num_samples), n_classes)
centers = X[rand_idx]
partial_dist = partial(dist, X)
    while True:
        # 2a. Assign data based on closest centre
        distances = np.array(list(map(partial_dist, centers)))
        clusters = np.argmin(distances, axis=0)

        # 2b. Update cluster centres from means of each cluster
        for i in range(n_classes):
            new_centers[i, :] = X[np.where(clusters == i)].mean(axis=0)
```

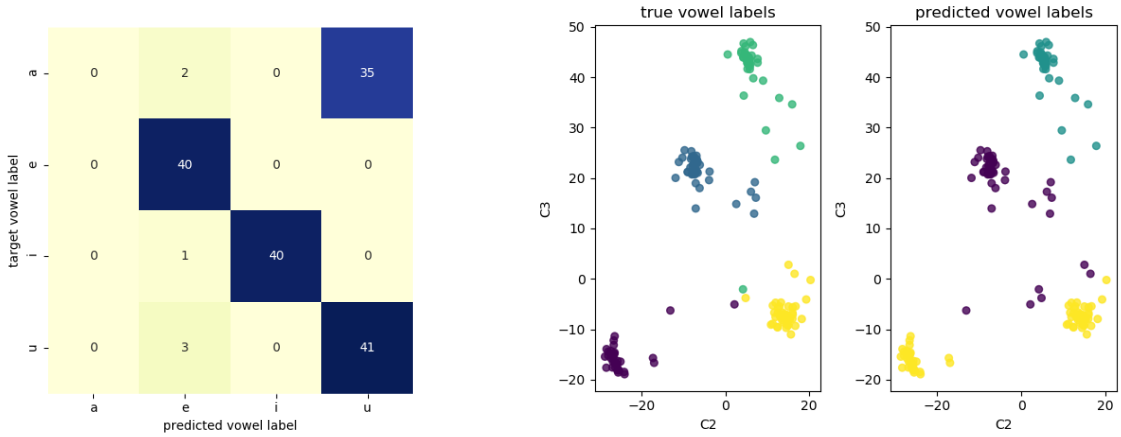

Figure 1: Example of poor results from Initial implementation

Better Initialization

A simple **k-means++** (Listing 2) initialization algorithm has been implemented where first center is picked at random and the following is picked with probability proportional to the distance from previous. Running evaluation confirms the effectiveness of this approach with mean F1 score = 0.91 and variance = 0.01. Although still not at a level of scikit-learn implementation it is at least comparable. The result could be further improved by running the algorithm multiple times and averaging the centroids.

Listing 2: Variation of k-means++ algorithm

```
rand_idx = []
# First choice is selected randomly with uniform probabilities
p = np.random.uniform(size=n_samples)
for _ in range(n_classes):
    r = rand_choice(p=p/np.sum(p))
    rand_idx.append(r)
    # Next probability is proportional to min distance squared:
    p = np.min(list(map(part_dist, X[rand_idx])), axis=0) ** 2
```

Feature Representations

At first, delta and delta delta MFCC's were calculated to encapsulate signal's temporal information. Initial experiments with those added demonstrated poor performance achieving at most F1 score of 0.85. Some investigation shown that this could be due to the difference in distributions between the standard MFCC's and deltas. Scaling all of the features increased the F1 score to 0.93 demonstrating a small improvement from using just MFCC's on their own.

Then, a different feature set has been computed in a form of Linear Prediction Coefficients (LPC). LPCs have been calculated from scratch for frames of 25ms with Hamming Window applied. First auto-correlation matrix **R** is computed and **A** coefficients are found using Levinson-Durbin algorithm. Initial experiments only achieved up to 0.65 F1 score using either personal implementation of **sckiti-learn**. This is likely due to incorrect distance metric used for sequential data, which will be further explored in the following section.

Distances

Both **Mahalanobis** and **Itakura** distances have been implemented as demonstrated in Listings 3 and 4 respectively. For Mahanalobis inverse covariance (**inv_cov**) is computed outside on the full dataset. Itakura distance is only used for LPC's where **R** stores auto-correlation matrix for each sample sequence in **X**. Results suggest Mahalanobis distance is not nearly as effective as Euclidean for this particular dataset, achieving F1 score of 0.7 for MFCC's and 0.35 for LPC's. This could likely be attributed to a non-normal nature of feature distribution particularly for LPC's. On the other hand, using Itakura distance for LPC's improved performance greatly achieving up to 0.85 F1 score.

Listing 3: Implementation of the Mahalanobis distance

```
def mahalanobis(X, y, inv_cov=None):
    if inv_cov is None:
        # Equivalent to euclidean
        inv_cov = np.eye(X.shape[1])
    diff = X - y
    d_2 = np.dot(np.dot(diff, inv_cov), diff.T)
    return d_2.diagonal()
```

Listing 4: Implementation of the Itakura distance

```
def itakura(a, b, R=None):
    n_samples = a.shape[0]
    dI = np.empty(n_samples)
    for i in range(n_samples):
        # Error for linear filter a and b:
        e_a = np.dot(np.dot(a[i], R[i]), a[i])
        e_b = np.dot(np.dot(y, R[i]), y)
        dI[i] = -np.log(e_a / e_b)
    return dI
```

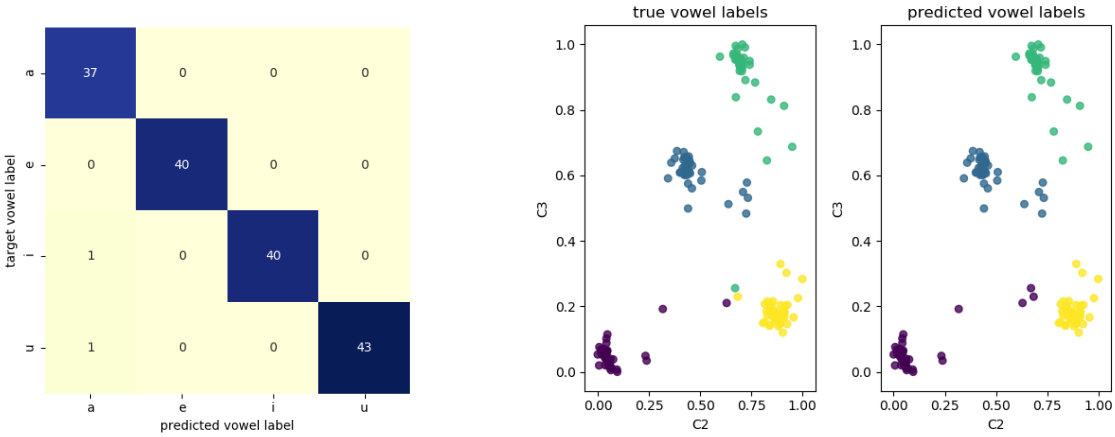

Figure 2: Top Results with Personal Implementation using MFCC's with deltas and Euclidean distance.