M4S18A2 Machine Learning - Coursework 2

Omar Haque

March 15, 2018

Introduction

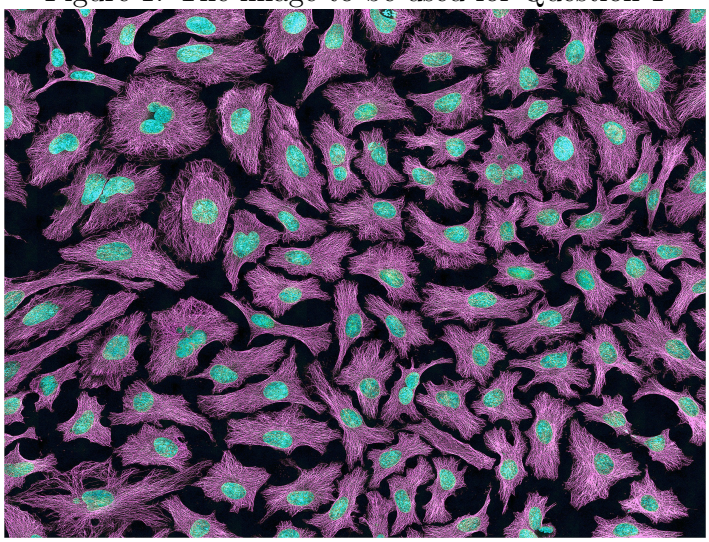
This report has 3 sections. In the first, I implement a K-means clustering algorithm, and a Gaussian Mixture Model under the Expectation-Maximisation framework. I use these algorithms to create a program which automatically segments and counts the number of cells in a medical image. In the second part I explore the role of bias in modern AI applications. In the third section, I implement a Hidden-Markov Model in order to create a predictive trading model for Ethereum prices.

Question 1

The Data

For this task I have a 1927x2560 pixel image of numerous cells, where each pixel holds a 3d RGB value. The task is to use K-Means clustering and Gaussian Mixture models to (a) produce an automatic segmentation of the image and then (b) automatically count the number of cells in the image.

Figure 1: The image to be used for Question 1



Algorithm Implementation

The K-Means algorithm is an unsupervised learning technique which groups N data points each into the cluster with the nearest mean. The basic details for the algorithm are below:

1. Initialise the k centroids (cluster means) by randomly sampling k points from the dataset. This is called the Forgy method of initialization.

2. Assignment: for each datapoint, assign it to the cluster whose mean is closest to it. Here is the function for this:

```
def get_closest_cluster(data, centroids,K):
    """
    Given the data and current centroids estimate, calculate the cluster for
    each datapoint in data
    Return data clustering
    """

    norms_matrix = np.zeros((data.shape[0],K))
    for k in range(K):
        norms_matrix[:,k] = np.linalg.norm(data - centroids[k],axis=1)
    data_clustering = np.argmin(norms_matrix,axis=1)

    return data_clustering
```

3. Update: Now that all datapoints have been assigned to a cluster, recompute the centroids for each of the K clusters. Here is the function for this:

```
def update_centroids(data, data_clustering,K):
    """
    Given the data and current data clusters, recompute the centroids for
    each cluster
    Return updated centroids
    """

    updated_centroids = np.zeros((K,data.shape[1]))

    for k in range(K):
        kth_points = data[data_clustering == k,:]
        centroid_kth = np.mean(kth_points,axis=0)
        updated_centroids[k] = centroid_kth

    return(updated_centroids)
```

4. If the convergence criteria has not occured (i.e, the centroids have moved significantly from the last iteration) GOTO 2. Else, END.

The algorithm in total using the functions above is below:

```
def k_means_clustering(X_cleaned,K=10,maxiter=200):

    # get dimensions of data
    n = X_cleaned.shape[0]
    dim = X_cleaned.shape[1]

    ## initialise centroids randomly
    random.seed(1)
```

```
    centroids = X_cleaned[random.sample(range(n),K),:]

    # initialise convergence criteria
    converged = False

    for i in range(maxiter):

        # bookkeeping, simply keep track of the old centroids
        old_centroids = centroids

        # data assignment step
        data_clustering = get_closest_cluster(X_cleaned, centroids,K)
        # centroids update step
        centroids = update_centroids(X_cleaned, data_clustering,K)

        if (np.allclose(centroids,old_centroids,rtol=1e-3)):
            print("converged at iteration " + str((i+1)))
            converged = True
            break

    if (not converged):
        print("Warning, clusters did not converge.")

    return centroids, data_clustering
```

I also implemented the Gaussian Mixture Model under the Expectation Maximization framework. The Gaussian Mixture model assumes that each datapoint x_n has the following pdf:

$$p(x_n) = \Sigma_{k=1}^{K} \pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k)$$

Formally, a latent variable z_{nk} is introduced and the "full" expected likelihood can be maximised, rather than the typical log likelihood which is prone to singularities in this case. Below are some of the basic details for the algorithm:

1. Initialise the parameters $\theta = \{\mu_k, \Sigma_k, \pi_k\}_{k=1:K}$. In my case, the default is to randomly cluster the data into K clusters, and then compute the corresponding means, variances and mixtures for those clusters. Code for this is here:

```
    ## Gaussian mixture models ##
    def initialise_parameters(X_cleaned,K=10):

        # initialise some key components
        random.seed(1)
        parameters = {}
        dim = X_cleaned.shape[1]
        n = X_cleaned.shape[0]

        # randomly split the dataset into K clusters.
        cluster_choices = np.array(random.choices(range(K),k=n))

        # initialise the arrays to be returned
```

```
    means = np.zeros((K,dim))
    covariances = np.zeros((K,dim,dim))
    mixtures = np.zeros(K)


    # go through this initial clustering and initialise the parameters
    for i in range(K):

        X_k = X_cleaned[cluster_choices == i,:]
        means[i,:] = np.mean(X_k,axis=0)
        covariances[i,:,:] = np.cov(X_k.transpose())
        mixtures[i] = X_k.shape[0]/n


    parameters["means"] = means
    parameters["covariances"] = covariances
    parameters["mixtures"] = mixtures


    return parameters
```

2. Compute the expected value of the latent variables, under the posterior and current estimate of parameters.

$$\gamma_{nk} := E[z_{nk}] = \frac{\pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k)}{\Sigma_{l=1}^{K} \pi_l \mathcal{N}(x_n|\mu_l, \Sigma_l)}$$

In fact the denominator is just a scaling factor which can be computed later by normalizing across rows. Below is the implementation:

```
def compute_expectations(X_cleaned,params,K=10):
    """
    :param X: The data matrix
    :param params: The current estimate for the parameters
    :param K: The number of mixtures
    """


    full_n = X_cleaned.shape[0]


    # initialise E, the matrix of expectations
    E = np.zeros((full_n, K))


    for k in range(K):
        # this is the numerator, the denominator is just a scaling factor
        E[:,k] = params["mixtures"][k] * multivariate_normal.pdf(
                x=X_cleaned,
                mean=params["means"][k],
                cov=params["covariances"][k])
```

```
    # normalise across rows
    E = E/E.sum(axis=1, keepdims=True)

    return E
```

3. Compute the estimates for the parameters which correspond to an increase in the log likelihood. These have been derived in lectures.

$$\pi_k = \frac{\Sigma_{n=1}^{N}\gamma_{nk}}{N},\ \mu_k = \frac{\Sigma_{n=1}^{N}\gamma_{nk}x_n}{\Sigma_{n=1}^{N}\gamma_{nl}},\ \Sigma_k = \frac{\Sigma_{n=1}^{N}\gamma_{nk}(x_n-\mu_k)(x_n-\mu_k)^T}{\Sigma_{n=1}^{N}\gamma_{nl}}$$

The code for this is below:

```
def maximisation_step(X_cleaned,expectations,K=10):

    # clean the data into an appropriate shape
    dim = X_cleaned.shape[1]
    full_n = X_cleaned.shape[0]

    # initialise params
    params = {}

    # initialise the arrays to be returned
    means = np.zeros((K,dim))
    covariances = np.zeros((K,dim,dim))
    mixtures = np.zeros(K)

    for k in range(K):
        # means
        means[k] = (X_cleaned.T * expectations[:,k]).sum(axis=1) / np.sum(expectations[:,
        # covariances
        Y = (X_cleaned - means[k]).T # scale the data by the mean and transpose
        Y = Y * np.sqrt(expectations[:,k])    # multiply by square root of responsibilit
        covariances[k] = np.matmul(Y,Y.T) / np.sum(expectations[:,k]) # perform the calcu
        # mixtures
        mixtures[k] = np.sum(expectations[:,k]) / float(full_n)

    params["means"] = means
    params["covariances"] = covariances
    params["mixtures"] = mixtures

    return params
```

The algorithm implementation is below:

```
def run_GMM(X_cleaned,K=10,params = -1,seed=1):

    if (params == -1):
        random.seed(seed)
        params = initialise_parameters(X_cleaned,K)

    for i in range(30):
```

```
        expectations = compute_expectations(X_cleaned,params,K)
        params = maximisation_step(X_cleaned,expectations,K)

    return params, expectations
```

Automated Counting Task

If we unpack the 1927x2560x3 3d image matrix into a 4933120x3 2d matrix we lose the pixel group structuring, but we can then apply the clustering/mixture algorithms described earlier. We can apply the algorithms to find the suitable collection of centroids/cluster means which then provides a segmentation if we replace each pixel intensity vector by the cluster mean/centroid vector as advised. For K-means the centroid vectors are an obvious choice, but for the Gaussian Mixture Model (GMM) we have to use the posteriors to then apply a hard classification rule (maximum likelihood across mixtures), upon which we can use the corresponding mixture means as the pixel vector.

Figures 2:7 show the results of the image segmentation process for both algorithms. As discussed in lectures, the K-Means algorithm is actually a special case of GMM EM in which the cluster covariance structure is assumed to be isotropic. Interestingly, for $K = 5$ both approaches yield similar images, but for $k = 3, 2$ we see that the GMM EM algorithm identifies the specific cell structure more strongly than K-Means. This is to be expected, as it encompasses a much larger variance structure.

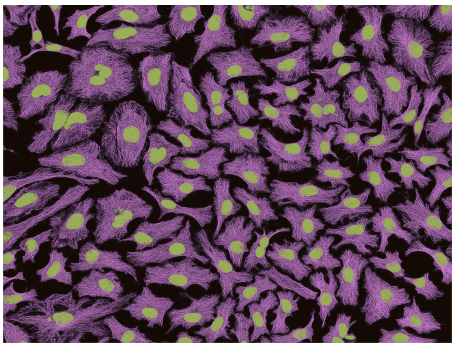


Figure 2: GMM segmentation with k = 5

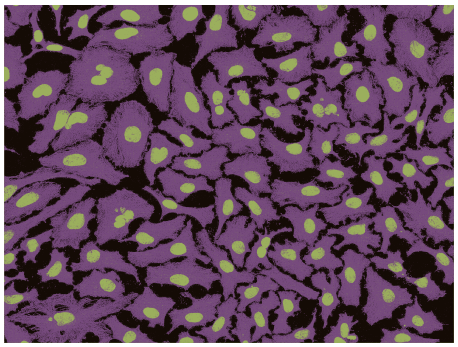


Figure 3: GMM segmentation with k = 3

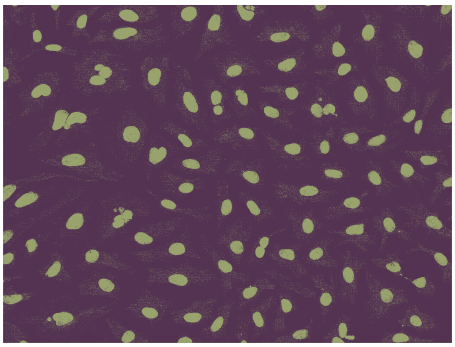


Figure 4: GMM segmentation with k = 2

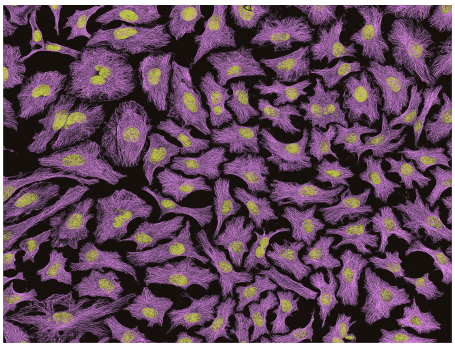


Figure 5: K-Means segmentation with k = 5

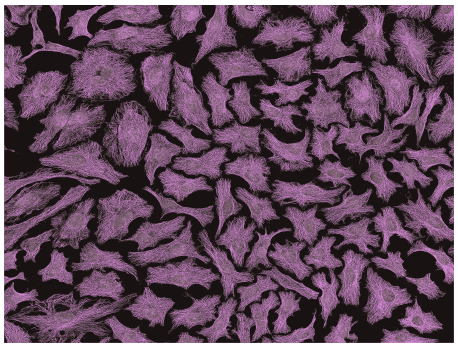


Figure 6: K-Means segmentation with k = 3

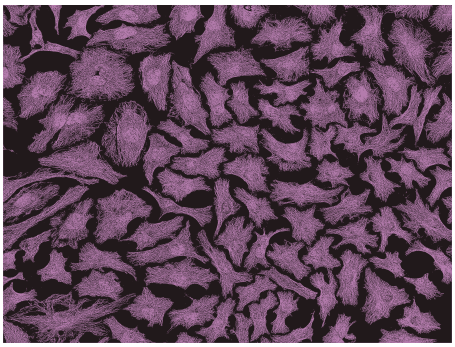


Figure 7: K-Means segmentation with k = 2

We now compute the cell counting process as follows:

1. Compute GMM Image segmentation with k = 2
2. Label points as "cell" or "not cell" depending on which cluster mean they belong to (what colour they are)
3. Extract only the points belonging to the correct cell colour (a 2d array of (x,y) values)
4. Perform k-means to initialise parameter values on this dataset
5. Perform GMM with different values of k. The value of k with the highest **To be filled** statistic will be the estimate for the number of cells.

Below is the code for this task

```
def count_cells(cleaned_dataset, cell_lower_bound, cell_upper_bound):
    """
    We use the AIC methodology described here:
        https://uk.mathworks.com/help/stats/tune-gaussian-mixture-models.html
    in order to find the number of cells, fine tuning the number of components
    in a gaussian mixture model. Can initialise with k means.
    """

    n = cleaned_dataset.shape[0]
    AIC_list = {}
    BIC_list = {}
    parameters = {}


    # initialise using kmeans
    centroids, data_clustering = k_means_clustering(cleaned_dataset,K=cell_lower_bound)
    # initialise the arrays to be returned
    means = centroids
    covariances = np.zeros((cell_lower_bound,2,2))
    mixtures = np.zeros(cell_lower_bound)


    # go through this initial clustering and initialise the parameters
    for i in range(cell_lower_bound):
        X_k = cleaned_dataset[data_clustering == i,:]
        covariances[i,:,:] = np.cov(X_k.transpose())
        mixtures[i] = X_k.shape[0]/n


    parameters["means"] = means
    parameters["covariances"] = covariances
    parameters["mixtures"] = mixtures


    for k_num in range(cell_lower_bound,cell_upper_bound+1):
        print("k is " + str(k_num))
```

```
        params, expectations = run_GMM(cleaned_dataset,K=k_num,params=parameters)
        log_lik = log_likelihood(cleaned_dataset,k_num,params,expectations)
        # compute AIC
        AIC_list[k_num] = -2*log_lik + 2 * k_num
        # compute BIC
        BIC_list[k_num] = -2*log_lik + k_num * np.log(n)

        parameters = params


    number_of_cells = np.argmax(BIC_list) + cell_lower_bound

    return number_of_cells

parameters, expectations = run_GMM(img_cleaned,K=3)
myimg_gmm = parameters["means"][np.argmax(expectations,axis=1)]
cell_colour = np.array([ 155.39698821,  175.87112501,   93.20177927])
myimg_gmm_blackwhite = np.where(np.isclose(myimg_gmm,cell_colour).all(axis=1),1,0).reshape((19
cleaned_dataset = np.column_stack(((np.where(myimg_gmm_blackwhite == 1))[0],np.where(myimg_gmm
count = count_cells(cleaned_dataset,82,90)
```

However, this does not work. Because of the large problem size the expectation values decrease exponentially and there's numeric underflow. I tried to work on the logscale but I do not now have enough time to figure out all the details.

Question 2

Intelligent Machines: Forget Killer Robots - Bias is the Real Danger in AI. - headline from a recent newspaper article.
Write up to 500 words discussing whether or not you agree with this headline regarding the imminent wide-spread application of AI and machine learning. You might, for example, focus on specific applications of machine learning and how biases might occur. Higher marks will be given for succinct, well thought through and well structured arguments. In particular, you should make reference to the concepts and methodologies presented in the lectures.

The worry of deadly machines and killer robots has permeated public culture for years. Yet even as we make huge strides in machine learning, most experts agree were still far from a robot uprising [1]. Here I explore the far bigger danger of bias and how it will have huge consequences with the current wide-spread application of AI.

Increasingly within the US, predictive algorithms are used to compute risk scores for legal defendants. These scores are then used in an advisory setting to inform fundamental decisions on defendant sentencing. One of the widest used programs of this sort is COMPAS, which takes various factors from a defendant and then classifies their risk of recidivism. An article from ProPublica [2] explored the results from COMPAS and found that Black defendants were twice as likely as white defendants to be misclassified as a higher risk of violent recidivism. This is a key example of machine learning bias, as the algorithm is biased against black individuals because of the data used to train it. This can happen in many forms, either through highly unbalanced datasets, or simply biased data collection methods.

This highly worrying example isnt to say that AI shouldnt be used for life changing decisions, as in fact the human sentencing process contains many inherent biases too. The danger however lies with the

fact that the biases here are hidden within a black box model. To the layman, these computational risk assessments offer a seemingly objective model of reality. The truth is however that statistical models aim only to provide simple approximations of reality. I believe the secretive nature of algorithms used are one of the main contributing factors to the danger of bias. The creators of COMPAS do not share specific calculations as it claims they are proprietary. One way to tackle this problem is to ensure that when AI applications are being used in the public domain theres complete transparency for the data being used to train the model, and ultimately for how the model itself works.

The issue of the gap in understanding between machine learning practitioners and the general public exacerbates the problem further. In fact, the entire scope of knowledge for a machine learning engineer is not yet well defined. As an example, take Andrew Ngs highly successful Coursera courses on Deep Learning. The modules are extremely practical and offer a great introduction to the subject. But if machine learning engineers arent encouraged to understand the underlying assumptions of these models, how will we expect the general public to?

The issue of bias is extremely worrying not only because of how easily hidden it is between ML practitioners, but also to the general public at large. In order to tackle the issue there needs to be more transparency in exactly how models are applied and which datasets are used to train them. This will ensure that there is more scrutiny over ML applications so as not to incur bias, an integral part of the scientific process.

Question 3

In this section, I use a **Hidden Markov Model** in order to predict hourly Ethereum prices.

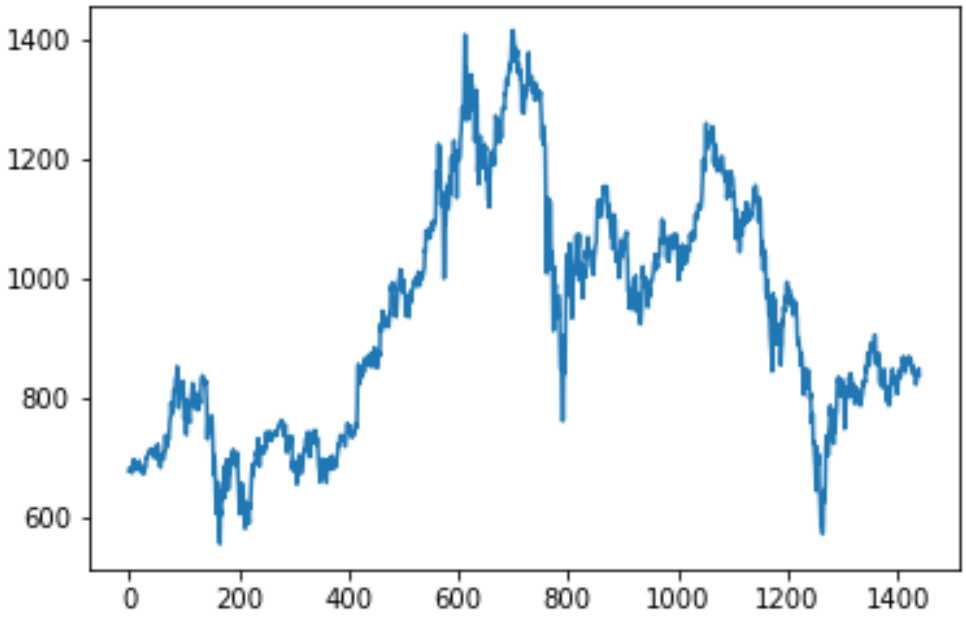


Figure 8: Plot of hourly Ethereum prices in the dataset

The dataset I'm considering has a number of different factors, however, for my model I'm only going to use the hourly open price, o_t (Figure 8), as I think it would be interesting to try and predict future prices without knowledge of spreads or volume, as a speculative investor may not on an exchange.

Firstly, at each hour t I calculate the difference between the open price at time $t+1$ and t, $i_t = o_{t+1} - o_t$, which you can see in Figure 9. I then threshold this according to whether it's increasing (1) or decreasing (0). So I have a series of 0s and 1s which I'm attempting to predict, e,g. [0,1,1,0,1,1,1,0,...].

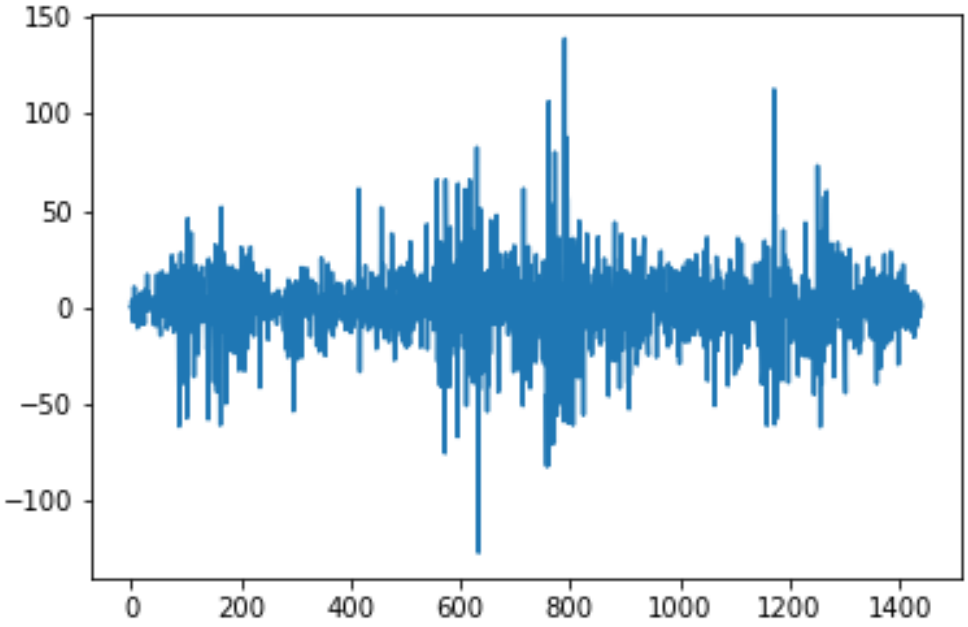


Figure 9: Plot of hourly price increments - the data to be modelled (after thresholding to 0s and 1s)

I've chosen to model this data since if we can use the data up until time t to predict i_t, then we can place small trades at each hour. The strategy is then as follows: go long on ethereum if we predict $i_t > 0$, and short if $i_t < 0$. This way, as long as we can get more movements correct than incorrect - this method is profitable.

Great care needs to be taken so only the data up until time t (inclusive) is used to predict i_t. Otherwise we encounter look ahead bias.

I chose the Multinomial Hidden Markov Model (MHMM) because I want to utilise the momentum relationship within the prices. If the increments are positive, they're likely to remain positive for a short while, and vice versa. The transition matrix between the latent variables models this perfectly. Also, MHMM takes only the previous latent variable into account. I think this will also give a good fit as the data looks quite noisy so the memoryless property of Markov Chains may be suitable here.

Model details

We follow closely the prescription given in Bishop [1]. Let x_n be our observations, each single time slice of the model corresponds to a mixture distribution with components given by $p(x_n|z_n)$ with discrete multinomial variables z_n describing which component of the mixture is responsible for generating the observation. We use a 1-of-K coding scheme for z_n. Let A represent the transition state space so that $A_{ij} = p(z_{nk} = 1|z_{n-1,j} = 1)$, let $\pi_k = p(z_{nk} = 1)$, and let the emission probabilities be represented by $b_t = p(x_t|z_t)$. Our parameter space that we need to estimate via Expectation Maximization is therefore $\theta = \{A, b, \pi\}$.

The expectation step requires some extra vectors.

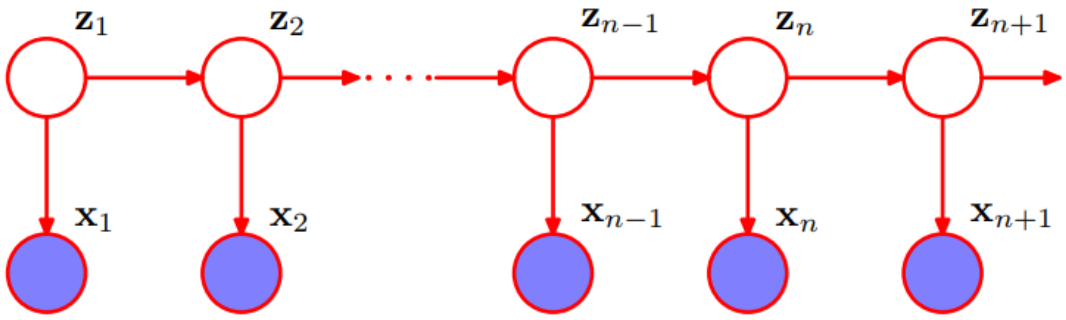


Figure 10: The graphical model for a HMM from Bishop [1]

Define the vectors α_t and β_t such that $\alpha_t(j) = p(z_{tj} = 1|x_{1:t})$ and $\beta_t(i) = p(x_{1:T|z_{t-1,i}=1})$. It's clear that $\gamma_t(j) := p(z_t = j|x_{1:T}) \propto \alpha_t(j)\beta_t(j)$. In fact since we only need this up to proportionality in z_t, we can find scalar multiples of α_t, β_t to stop underflow.

The trick here is that we can use the product rule, sum rule and dynamic programming algorithms to efficiently calculate $\alpha t and \beta_t$. The proof of the Forward Algorithm is here [5].

Forward Algorithm

1. Input: A, b_t, π
2. $[\alpha_1, Z_1] = normalize(b_1\pi)$
3. **for** $t = 2 : T$ **do**
4. $[\alpha_t, Z_t] = normalize(b_t(A^T\alpha_{t1}))$;
5. Return $\alpha_{1:T} and log[p(x_{1:T})] = log[Z_t]$
6. Subroutine: $[v, Z] = normalize(u) : Z = u_j; u_j = u_j/Z$;

Here is the code for the forward filtering algorithm [1]

```
def ForwardFiltering(A, b, pi, N, T):
    """Filtering using the forward algorithm (Section 17.4.2 of K. Murphy's book)
    Input:
      - A: estimated transition matrix
      - b: estimated observation probabilities (local evidence vector)
      - pi: initial state distribution pi(j) = p(z_1 = j)
      - N: number of hidden states
      - T: length of the sequence

    Output:
      - Filtered belief state at time t: alpha = p(z_t|x_1:t)
      - log p(x_1:T)
      - Z: normalization constant"""

    # initialise alpha, Z
```

[1] I'd like to note that the structure for the Python code throughout this section was given as an assignment in CO495. All functions were empty, and I have filled everything out myself and referenced where required, but the actual barebones structure for how they fit together was given in that course.

```
    alpha = np.zeros((N,T))
    Z = np.zeros((T))

    # The below algorithm is given exactly by the pdf notes

    alpha[:,0], Z[0] = normalize(b[:,0] * pi[:,0],dim=0)

    for t in range(1,T):
        alpha[:,t], Z[t] = normalize(b[:,t] * np.matmul(np.transpose(A),alpha[:,t-1] ))

    # definition of the logProb
    logProb = np.sum(np.log(Z))

    return alpha, logProb, Z
```

We also need the backward algorithm [6] for the β's: **Backward Algorithm**

1. Input: A, b_t
2. $beta_T = [1, ..., 1]$
3. **for** $t = T - 1, ..., 1$ **do**
4. $\beta_{t1} = A(b_t \beta_t)$
5. Return $\beta_{1:T}$

Again, we only need $\alpha * \beta$ up to proportionality in z_t, so I compute a scaled version of β by the log likelihood to prevent underflow.

```
def BackwardFiltering(A, b, N, T, Z):
    """Perform backward filtering.
        Input parameters:
            - A: estimated transition matrix (between states)
            - b: estimated observation probabilities (local evidence vector)
            - N: number of hidden states
            - T: length of the sequence
        Output:
            - beta: filtered probabilities
    """

    # initialise the shape of the entire beta matrix
    beta_scaled = np.zeros((N,T))
    # set beta_T
    beta_scaled[:,T-1] = np.array([1,1])

    # iterate backwards and find beta_t for t = 1,...,T-1
    for t in reversed(range(0,T-1)):
        beta_scaled[:,t] = (A @ (b[:,t+1] * beta_scaled[:,t+1]))/Z[t+1]

    return beta_scaled
```

The full forward-backward algorithm is given by simply combining the two

```
def ForwardBackwardSmoothing(A, b, pi, N, T):
    """Smoothing using the forward-backward algorithm.
    Input:
      - A: estimated transition matrix
      - b: local evidence vector (observation probabilities)
      - pi: initial distribution of states
      - N: number of hidden states
      - T: length of the sequence
    Output:
      - alpha: filtered belief state as defined in ForwardFiltering
      - beta: conditional likelihood of future evidence as defined in BackwardFiltering
      - gamma: gamma_t(j) proportional to alpha_t(j) * beta_t(j)
      - lp: log probability defined in ForwardFiltering
      - Z: constant defined in ForwardFiltering"""

    # forward filter, then backward filter, then normalize.
    alpha, logProb, Z = ForwardFiltering(A,b,pi,N,T)
    beta_scaled = BackwardFiltering(A,b,N,T,Z)
    gamma, norm = normalize(alpha * beta_scaled,dim=0)

    return alpha, beta_scaled, gamma, logProb, Z
```

And for the final part of the expectation stage, it turns out that we can also break down the two slice posterior into new vectors $\epsilon_{t,t+1} := p(z_t, z_{t+1}|x_{1:T}) \propto p(z_t|x_{1:T})p(x_{t+1:T}|z_t, z_{t+1})p(z_{t+1}|z_t) = p(z_t|x_{1:T})p(x_{t+1}|z_{t+1})p(x_{t+2:}$ $A \circ (\alpha_t(b_{t+1} \circ \beta_{t+1})^T)$

```
def SmoothedMarginals(A, b, alpha, beta_scaled, T, Nhidden, Z):
    "Two-sliced smoothed marginals p(z_t = i, z_t+1 = j | x_1:T)"
    marginal = np.zeros((Nhidden, Nhidden, T-1));
    for t in range(T-1):
        marginal[:, :, t] = (normalize(A * np.outer(alpha[:, t], np.transpose( (b[:, t+1] * be

    return marginal
```

Finally for the maximization step we need to find the maxima for the full joint log likelihood, $ln(p(X, Z|\theta)) = \Sigma_{t=1}^{T}\Sigma_{j=2}^{M}\Sigma_{k=1}^{K}x_{tj}E[z_{tk}]ln(b_{jk}) + \Sigma_{k=1}^{K}E[z_{1k}]ln(\pi_k) + \Sigma_{t=2}^{T}\Sigma_{j=1}^{2}\Sigma_{k=1}^{K}E[z_{t-1,j}z_{tk}]ln(a_{jk})$, applying the relevant lagrange multipliers for each parameter we arrive at:

$$\pi_k = \frac{E[z_{1k}]}{\Sigma_{r=1}^{K}E[z_{1r}]}, b_{jk} = \frac{\Sigma_{t=1}^{T}E[z_{tk}]x_{tj}}{\Sigma_{t=1}^{T}E[z_{tk}]}, A_{jk} = \frac{\Sigma_{t=2}^{T}E[z_{t-1,j}z_{t,k}]}{\Sigma_{r=1}^{K}\Sigma_{t=2}^{T}E[z_{t-1,j}z_{t}r]}$$

.

The entire expectation-maximisation algorithm is therefore given by the following code:

```
def EM_estimate_multinomial(Y, Nhidden, Niter, epsilon, init):

    # Dimensions of the data
    N, T = Y.shape

    # extract initialisations
    A = init["A"]
    B = init["B"]
```

```
pi = init["pi"]

##############################################
# EM algorithm

i = 0
# Initialize convergence criteria
logProbOld = -100000
logProbDiff = epsilon + 1
while ((i<Niter)and((logProbDiff > epsilon))): # and condition on criterion and precision
    # Iterate here
    # EXPECTATION:
    # iterate through all the sequences, and compute the responsibilities and smoothed m
    b = computeSmallB_Discrete(Y[0], B)
    ### expectation step
    alpha, beta, gamma, logProbNew, Z = ForwardBackwardSmoothing(A, b, pi, Nhidden, T)
    epsilon = SmoothedMarginals(A, b, alpha, beta, T, Nhidden,Z)

    # MAXIMISATION:
    # compute pi, A as per the before formulas.
    for k in range(Nhidden):
        pi[k] = gamma[k,0] / np.sum(gamma[:,0])
        for j in range(Nhidden):
            A[j,k] = np.sum(epsilon[j,k,:])

    # normalise A
    A,_ = normalize(A,dim=1)
    # maximisation step for B:
    # use the suggested one hot encoding and then the formula derived.
    Bnew = np.zeros((Nhidden,2))
    for l in range(N):
        X = np.zeros((T,2))
        for m in range(T):
            X[m,Y[l,m]] =1
        Bnew = Bnew + np.matmul(gamma,X)

    B,_ = normalize(Bnew,dim=1) # normalise B across rows.
    # update convergence criteria
    logProbDiff = logProbNew - logProbOld
    logProbOld = logProbNew
    i+=1
```

Test Procedure

In order to actually predict future results I use the following scheme:

Let θ_t be the EM estimates for the parameters using data up to t: $o_1, .., o_t$.

We can uncover $ln(P(o_t + 1|o_1, ..., o_t; \theta_t))$ directly from the forward filtering algorithm (Z_{T+1}). This means I will set as the prediction for o_{t+1}, the value which gives the highest log probability, conditioned on the information up to time t. This is a hard assignment based on the conditional.

```
def multiHMMtest(data_with_thresholding,train_split=200):


    train = data_with_thresholding["difference_threshold"][0:train_split]
    train = train.reshape(1,train_split)
    # initialise parameters somehow
    init = {}

    A = np.array([[0.4,0.6],[0.45,0.55]])
    B = np.array([[0.4,0.6],[0.45,0.55]])
    pi = np.array([[0.45],[0.55]])

    init["A"] = A
    init["B"] = B
    init["pi"] = pi

    A,B,pi = EM_estimate_multinomial(train, 2, 100, 0.1, init)


    model_predictions = []

    for i in range(train_split,1440):
        # predict 201th to 1440th values

        # initialise with last values
        init = {}
        init["A"] = A
        init["B"] = B
        init["pi"] = pi

        # perform EM on all previous values (not including i)
        A,B,pi= EM_estimate_multinomial(data_with_thresholding["difference_threshold"][0:i].re
                                        2, 100, 0.1, init)

        Ydummy0 = np.append(data_with_thresholding["difference_threshold"][0:i].reshape(1,-1)[
        Ydummy1 = np.append(data_with_thresholding["difference_threshold"][0:i].reshape(1,-1)[

        b0 = computeSmallB_Discrete(Ydummy0[0], B)
        b1 = computeSmallB_Discrete(Ydummy1[0], B)


        alpha0, logProb0, Z0 = ForwardFiltering(A,b0,pi,2,i+1)
        alpha1, logProb1, Z1 = ForwardFiltering(A,b1,pi,2,i+1)


        # make prediction based on whichever logProb is highest.
        predictions = np.argmax(np.array([Z0[len(Z0)-1],Z1[len(Z1)-1]]))

        model_predictions.append(predictions)
    return model_predictions, A,B,pi
```

Results

I tested the predictions for this model on a 50-50 training/test split.
The model achieved a **52.78%** accuracy. Although this seems a mildly high score since the prediction are based simply on past prices alone, the confusion matrix shows that the model was not actually a good fit. See figure 11.

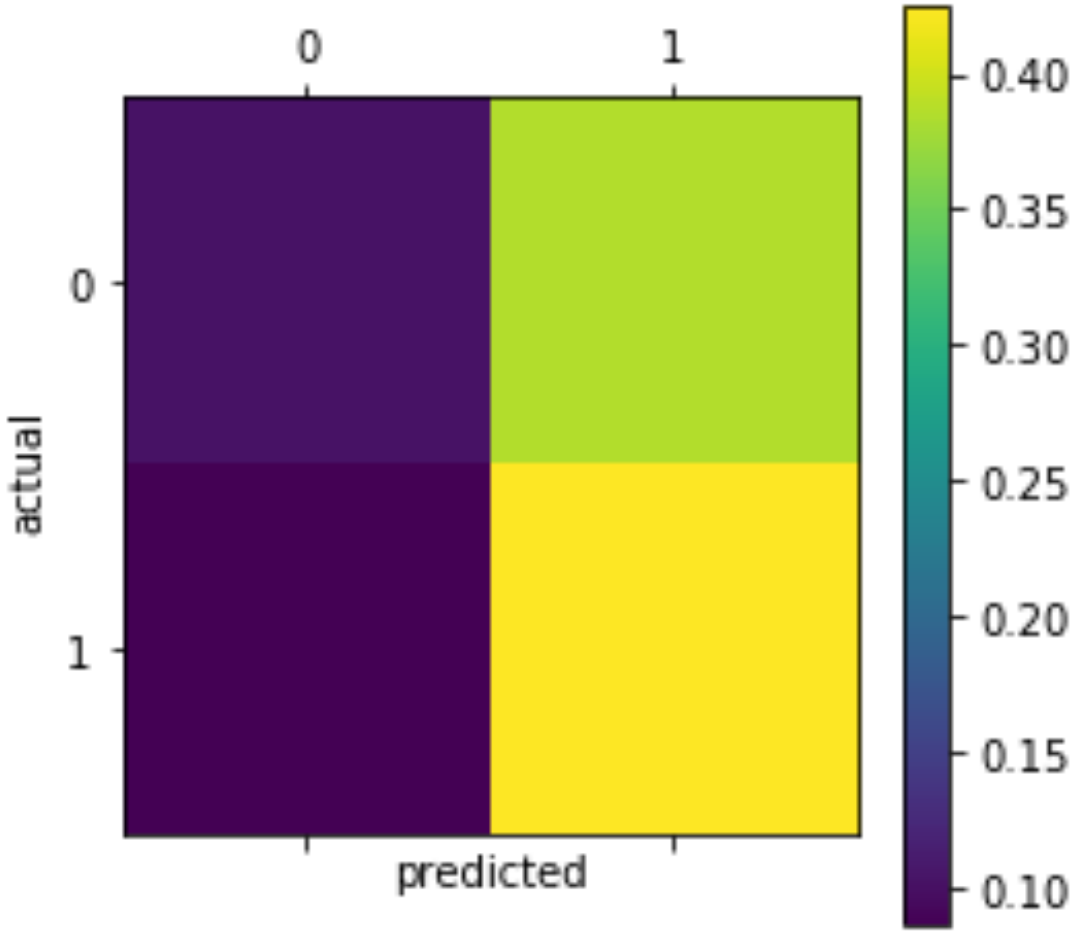


Figure 11: Heatmap for the confusion matrix

The model actually predicts a 1 approximately 80% of the time, even though there's a fairly even split between 0s and 1s. throughout the dataset. We look at the final parameter estimates for further analysis.

$$\mathbf{A} = \begin{pmatrix} 0.455 & 0.545 \\ 0.556 & 0.444 \end{pmatrix}, \ \mathbf{B} = \begin{pmatrix} 0.000 & 1.000 \\ 0.947 & 0.053 \end{pmatrix}, \pi = \begin{bmatrix} 1.00 \\ 0.00 \end{bmatrix}$$

The transition matrix and emission probabilities actually picked up exactly what I wanted it to. **A** shows that movement between the "up" and "down" states is considerably balanced, but given z is in one state it's slightly more likely to be in the other state next.
The matrix of emission probabilities B picked up exactly that the two latent variables are indeed the up/down classifiers.
But the prior is 1 for one component. This indicates to me that the Multinomial HMM was not a good fit for this process. In hindsight I should have perhaps used a simpler, more flexible model that might not necessarily take the sequenced data into account. Also, the multinomial distribution was not actually a good estimate for how the increments behave. Perhaps a Gaussian process would have been better.

I would not use this model to actually trade. As I've described, the multinomialHMM was not a good predictor for future prices. Even though the accuracy was slightly above 50%, the confusion matrix shows that there were many false positives. See figure 12 for the results of the prediction, they only started flicking between the two scores at a very late stage.

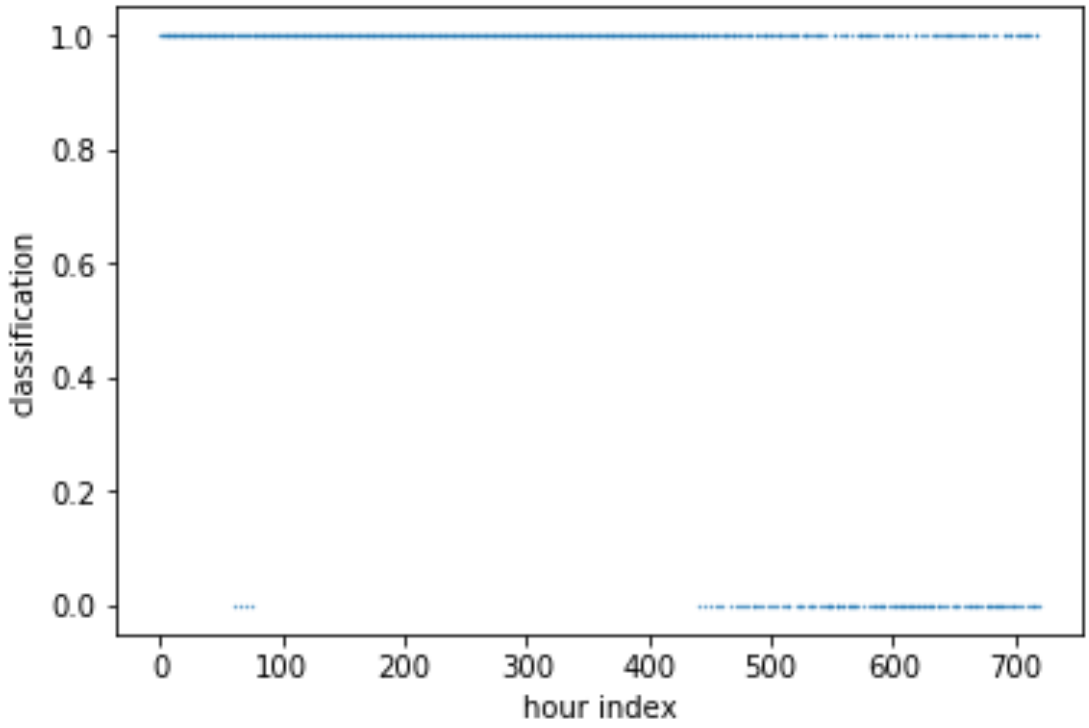


Figure 12: Predicted values for the MHMM

1 Appendix

A Question 1 - imports and utility functions

```
# imports
import cv2
import random
import numpy as np
from scipy.stats import multivariate_normal
import matplotlib.pyplot as plt

## preliminary functions ##

def clean_data(unclean_data):
    dim = unclean_data.shape[2]
    n = unclean_data.shape[0]*unclean_data.shape[1]
    data_cleaned = np.reshape(unclean_data,(n,dim))
    return data_cleaned
    img = cv2.imread("/Users/Omar/Documents/Year4/machineLearning/coursework2/" +
                "data/question1/FluorescentCells.jpg")
```

```
# img_cleaned is the data that is used throughout
img_cleaned = clean_data(img)
```

B Imports and Utility Functions for Question 3

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
# this data is the one referenced and used throughout
data = pd.read_csv("/Users/Omar/Documents/Year4/machineLearning/coursework2/data/question3/Eth

data["Difference"] = np.zeros(1440)

for i in range(1,1439):
    data.set_value(i,"Difference",data["Close"][i + 1] -data["Close"][i])


    test = np.where(data["Difference"] >= 0,1,0)
    data["difference_threshold"] = test

def normalize(A, dim=None, precision=1e-20):
    """
    The sklearn.preprocessing.normalize function was quite annoying to deal with,
    so I'm using this function which is adapted from Kevin Murphy's code for
    Machine Learning: a Probabilistic Perspective. It's just a normalization
    function.

    Make the entries of a (multidimensional) array sum to 1
    A, z = normalize(A) normalize the whole array, where z is the normalizing constant
    A, z = normalize(A, dim)
    If dim is specified, we normalize the specified dimension only.
    dim=0 means each column sums to one
    dim=1 means each row sums to one
    Set any zeros to one before dividing.
    This is valid, since s=0 iff all A(i)=0, so
    we will get 0/1=0
    Adapted from https://github.com/probml/pmtk3"""

    if dim is not None and dim > 1:
        raise ValueError("Normalize doesn't support more than two dimensions.")

    z = A.sum(dim)
    # If z is a scalar, z.shape is an empty tuple and evaluates to False
    if z.shape:
        z[np.abs(z) < precision] = 1
    elif np.abs(z) < precision:
        return 0, 1

    if dim == 1:
```

```
        return np.transpose(A.T / z), z
    else:
        return A / z, z

def computeSmallB_Discrete(Y, B):
    """Compute the probabilities for the data points Y for a multinomial observation model
        with observation matrix B

        Input parameters:
            - Y: the data
            - B: matrix of observation probabilities
        Output:
            - b: vector of observation probabilities
    """
    # initialise variables
    Nhidden = B.shape[0]
    T = len(Y)
    b = np.zeros((Nhidden,T))

    # simply select the appropriate values from the emission matrix B
    b[:,:] = B[:,Y[:]]

    return b
```

C Testing and analysis for Question 3

```
def analyse(actual,predicted):


    # accuracy score
    accuracy = 100*np.sum(np.equal(predicted,actual))/(len(actual))
    print(accuracy)
    # produce a heatmap/confusion matrix
    conf = np.zeros((2,2))

    for i in range(len(predicted)):
        conf[actual[i]][predicted[i]]+=1

    plt.matshow(conf/len(actual))
    plt.colorbar()
    plt.xlabel("predicted")
    plt.ylabel("actual")
    plt.show()

    plt.scatter(range(len(predicted)),predicted,s=0.1)
    plt.xlabel("hour index")
    plt.ylabel("classification")

    plt.show()
```

```
predictions1,predictions2,A,B,pi = multiHMMtest(data,720)
print(A,B,pi)
analyse(actual,predictions1)
```

References

[1] C. Bishop, Pattern Recognition and Machine Learning, (Springer,2006)