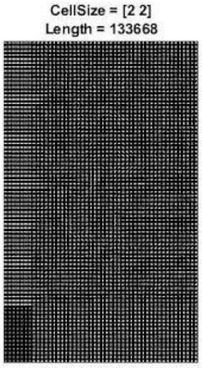
CSC3067 Group Assignment 2

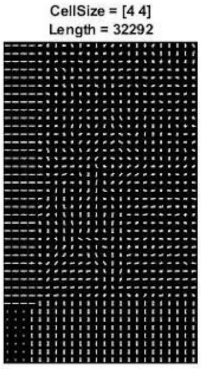
By Benjamin Grainger, Adrian Salondaguit and James Cassidy

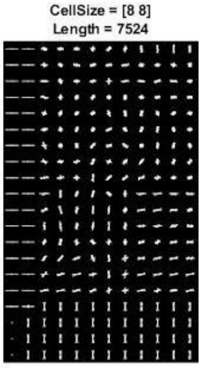
Feature Descriptor – Histogram of Oriented Gradients

Histogram of Oriented Gradients (HOG) is a feature descriptor that is used to extract features from image data. It is widely used for object detection and so will help in detecting pedestrians for this project.

When picking a cell size to use to obtain HOG features, we settled on [8, 8]. Below shows an image and its different HOG plots when used with [2,2], [4,4] and [8, 8] from left to right. The Computer Vision System Toolbox was used for extractHOGFeatures function.



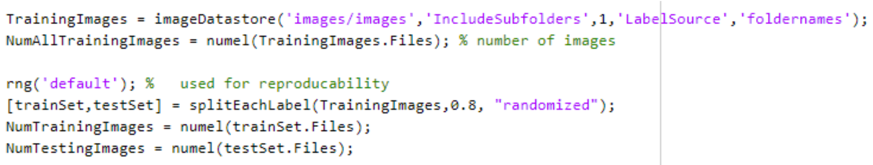




[2, 2] and [4, 4] encode lots of the shape of the image selected, but at a cost of increasing HOG feature vector size significantly. [8, 8] is a good middle ground as it encodes enough spatial information whilst also limiting the HOG Feature vector to 7524. This also speeds up processing time and reduces the file size of the model for later use in sliding windows and NMS.

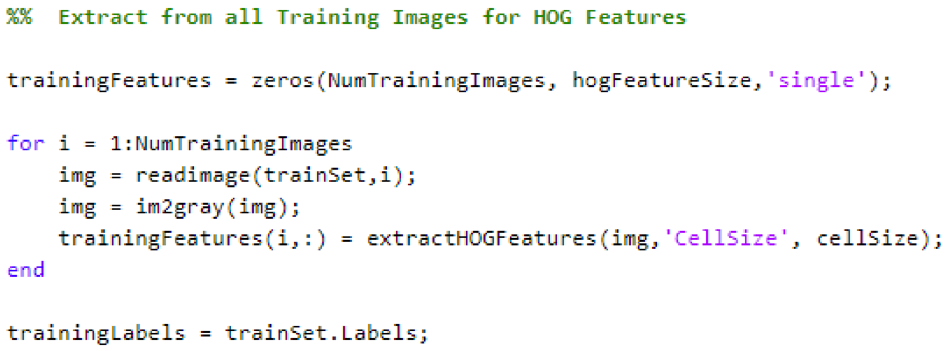
HOG SVM Classifier

When setting up the testing and training sets, we used a stratified 80:20 split. 80% of the positive and negative were used in the training set. The last 20% was used for testing. The use of the rng function was used for reproducibility when the program is run several times and ensures that there are both positive and negative images in both the training and test set. Had this not been used, the first 1000 images of the training set would have been negative images that do not contain any people. There are only 1000 negative images and therefore would leave none for testing. The testing image set is unseen by the model. To ensure the model is accurate, it must be tested against unseen data. The training set contains 2400 images, and the testing set contains 601.



```
TrainingImages = imageDatastore('images/images','IncludeSubfolders',1,'LabelSource','foldernames');
NumAllTrainingImages = numel(TrainingImages.Files); % number of images

rng('default'); %   used for reproducability
[trainSet,testSet] = splitEachLabel(TrainingImages,0.8, "randomized");
NumTrainingImages = numel(trainSet.Files);
NumTestingImages = numel(testSet.Files);
```

To extract the HOG feature vectors from the Training set, a for loop is run that applies pre-processing to all 2400 images by converting them to grayscale. After each image is pre-processed, the training features used will be extracted from the image with a cell size of [8,8] used, defined in a variable in the script. Once this has run, the training labels will be set to the labels used in the training imagestore.

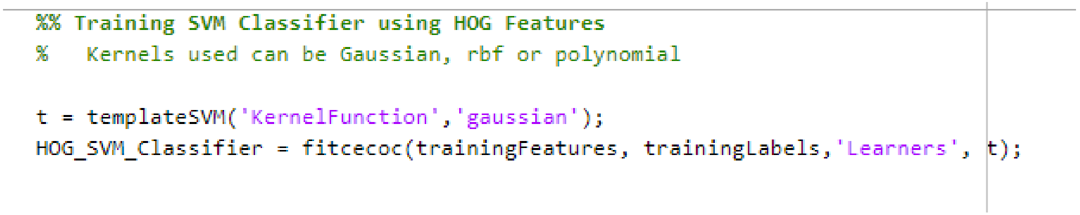


```
%%  Extract from all Training Images for HOG Features

trainingFeatures = zeros(NumTrainingImages, hogFeatureSize,'single');

for i = 1:NumTrainingImages
    img = readimage(trainSet,i);
    img = im2gray(img);
    trainingFeatures(i,:) = extractHOGFeatures(img,'CellSize', cellSize);
end

trainingLabels = trainSet.Labels;
```

Now that each image has had its HOG features extracted, we can apply Single vector machines (SVM) to the extracted HOG features. This is done using the fitcecoc function that is included in the Statistics Toolbox. The parameters of this function include the training features that have already been extracted from the training set, and the training labels that have also been obtained from the training set.

We also used other parameters such as ‘KernelFunction’. This was done within the templateSVM function and called using ‘Learners’ in fitcecoc. KernelFunction can be changed by the user of the script if they wish to see the results of a different kernel.

Standardize is also used within templateSVM. Standardizing the is essential for data analysis.



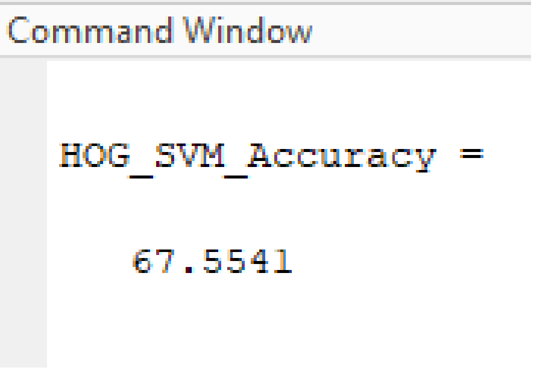
```
%% Training SVM Classifier using HOG Features
%   Kernels used can be Gaussian, rbf or polynomial

t = templateSVM('KernelFunction','gaussian');
HOG_SVM_Classifier = fitcecoc(trainingFeatures, trainingLabels,'Learners', t);
```

When applying a kernel function to the model, we used Gaussian, RBF and Polynomial. With the Gaussian and RBF, the accuracy returned was 67.05%. When changing the kernel to Polynomial, accuracy increased drastically to 97.67%. The confusion matrix that follows this will result in fewer false positives and false positives as a result. Therefore, when moving forward with this project, the polynomial kernel will be used for its increase in accuracy.

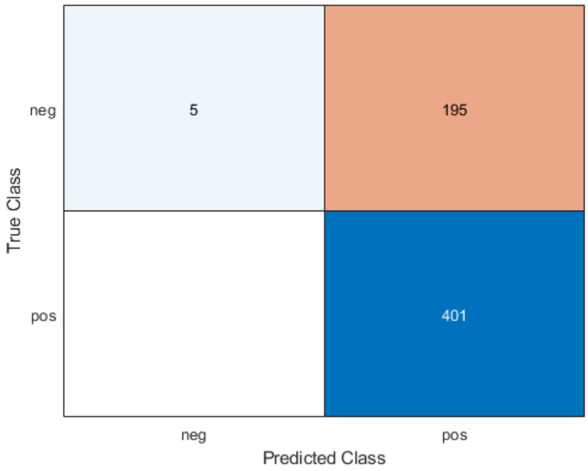
Looking at the Gaussian confusion matrix, we can see that it correctly predicted all 401 positive images. However, it also predicted that there were 195 images that were positive, while in fact they were negative. It only correctly predicted 5 negative images.

When we look at the polynomial confusion matrix, it paints a different story. While it predicted 399 out of 401 positive images correctly, using this kernel made a massive improvement in correctly predicting negative images with 188 out of 200 correctly predicted. This tells us that the HOG SVM Polynomial model can almost always tell whether there are or aren't pedestrians in an image, compared to Gaussian that almost always finds a person image when in fact a person does not exist.

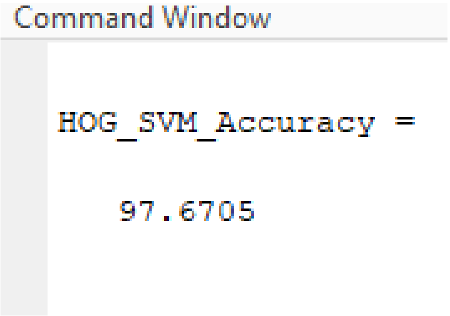
Gaussian Kernel:



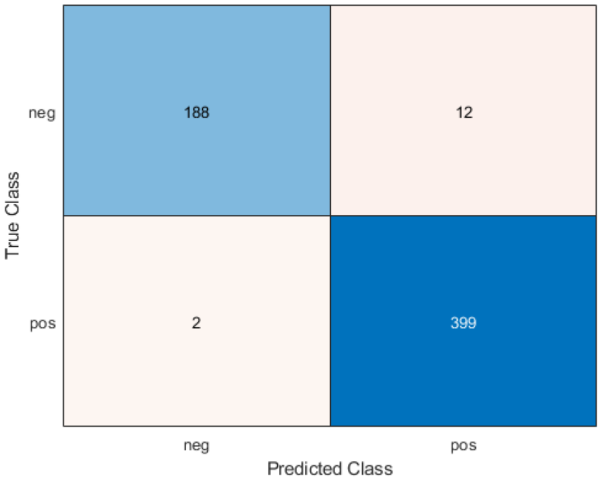
```
HOG_SVM_Accuracy =

   67.5541
```



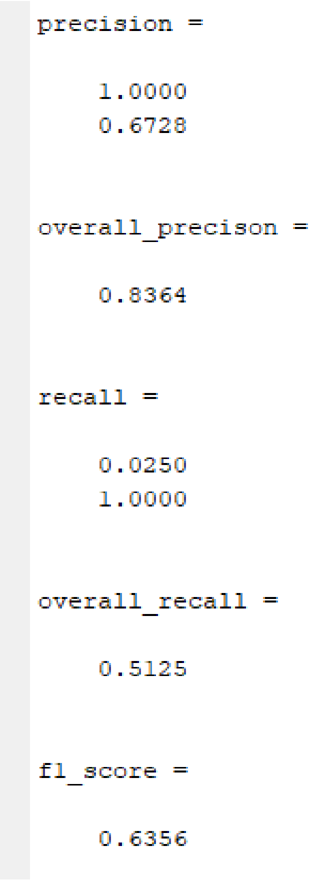
Polynomial Kernel:



```
HOG_SVM_Accuracy =

   97.6705
```

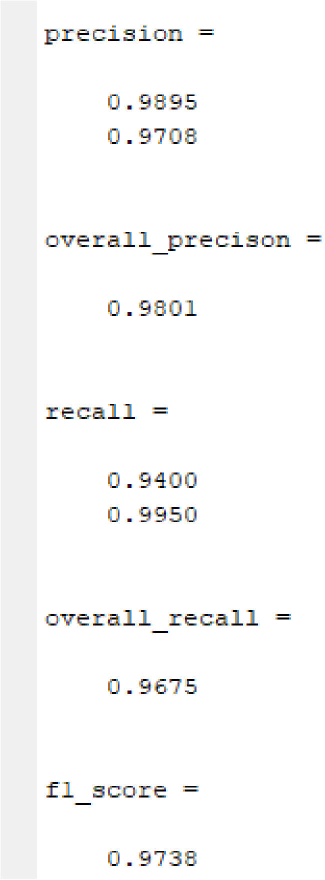


Gaussian Scores



```
precision =

    1.0000
    0.6728


overall_precison =

    0.8364


recall =

    0.0250
    1.0000


overall_recall =

    0.5125


f1_score =

    0.6356
```

Polynomial Scores



```
precision =

    0.9895
    0.9708


overall_precison =

    0.9801


recall =

    0.9400
    0.9950


overall_recall =

    0.9675


f1_score =

    0.9738
```

We can also evaluate the performance of these models' using precision, recall and F1 scores. As we already have the confusion matrix for both kernels this should be straight forward.

The precision score will be the ratio of correctly predicted positive results to the total amount of positive observations. A higher precision score relates to a lower false positive rate. In the case of Gaussian, we see an overall precision score of 0.8364. While this is quite a good score, we see for the Polynomial precision the score achieved was 0.9801. This tells us that the polynomial kernel gets a lower false positive rate than the gaussian kernel which is what we want.

Recall (or Sensitivity) is the ratio of number of correctly predicted positive observations to all the observations used in the confusion matrix. If the recall score is above 0.5 then this is considered a good score. For the Gaussian Kernel we see an overall recall of 0.5125 compared to a recall of 0.9675 for the polynomial kernel. The almost perfect recall of the polynomial tells us that it has a higher ratio in predicting the correct positive and negative images from our unseen data set compared to gaussian.

Finally, the F1 score takes the weighted average of both the precision and recall scores. Therefore, it considers the false positives and false negatives, making it more useful compared to the accuracy we calculated. Calculating the f1 score, we see for the Gaussian Kernel the score is 0.6356 while the f1 score for Polynomial is 0.9738. This is to be expected from how different the precision and recall scores were between the two models.

With these scores calculated, I can see that using the polynomial kernel for our feature descriptor is the only option to use for Sliding Window as the scores are almost perfect when compared to Gaussian. When compared to Full Image and PCA, HOG Polynomial is still the most accurate.

HOG KNN Classifier

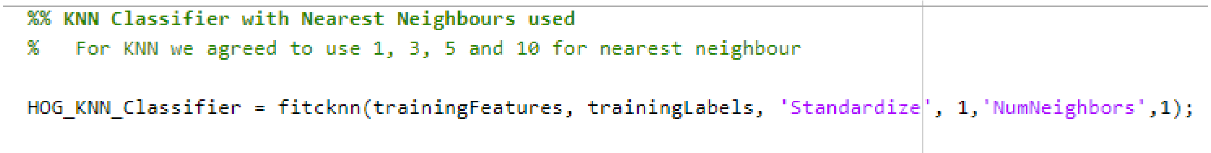
As there is no physical way to determine the best value for ‘K’ nearest neighbour, we have each set K to equal 1,3,5 and 10. We tried these values before settling on a value for ‘k’. Low values for K like 1 or 3 may be noisy and can be subject to outliers while large values can smooth things out so that a category only has a few samples. We kept this in mind as we tested with HOG.

We used the same split again for HOG KNN; a stratified 80:20 split. 80% of the images are used for training the model while 20% is used for testing and unseen. The 3000 images were randomised again using the splitEachLabel function and rng function, the latter ensures that the tests are reproducible with different values for K, as the randomised imagedatastore will always have the same random images.

HOG cell size of [8,8] was used again with KNN for its reduced runtime and file size and little loss in feature vectors.

All testing images were pre-processed to grayscale within and for loop and the results were saved within trainingFeatures. After these HOG features have been taken, the training labels are taken directly from the training set and are given the variable name ‘trainingLabels’

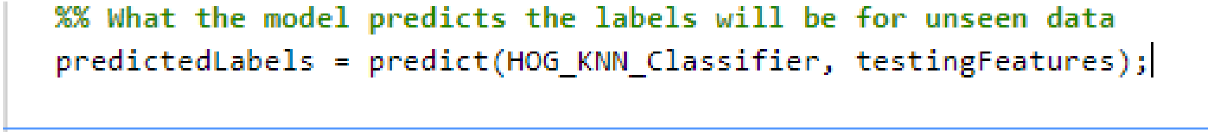
The classifier is saved using the fitcknn function. Standardize is used again. Also, within the function I can define the value of ‘k’ I wish to use for the classifier. This is done using ‘NumNeighbours’, followed by the value we wish to use for ‘k’. This value can be edited to whatever value the user wishes.



```
%% KNN Classifier with Nearest Neighbours used
%   For KNN we agreed to use 1, 3, 5 and 10 for nearest neighbour

HOG_KNN_Classifier = fitcknn(trainingFeatures, trainingLabels, 'Standardize', 1,'NumNeighbors',1);
```

Once the classifier has been saved, we then process the testing/unseen images by converting them to grayscale and gathering their HOG features. This will be stored within ‘testingFeatures’. The test labels used will be taken from the test set and be named ‘testingLabels’.

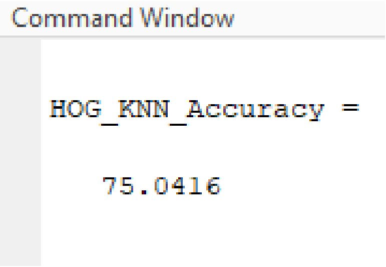
We can now use the predict function that will return a vector of the predicted class labels for the predictor data. In this case predict the labels of the ‘testingFeatures’ using the KNN classifier we have trained.



```
%% What the model predicts the labels will be for unseen data
predictedLabels = predict(HOG_KNN_Classifier, testingFeatures);
```

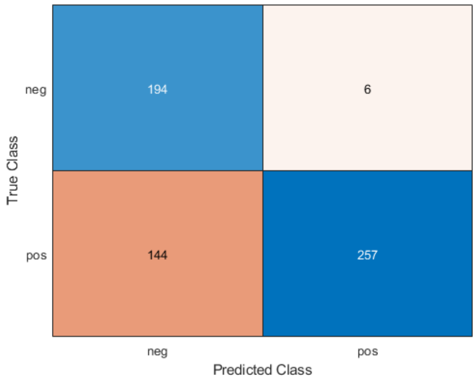
We can now find out the accuracy for each value of K that we used.

K = 1

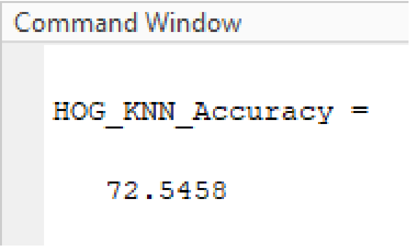


Command Window

```
HOG_KNN_Accuracy =

   75.0416
```

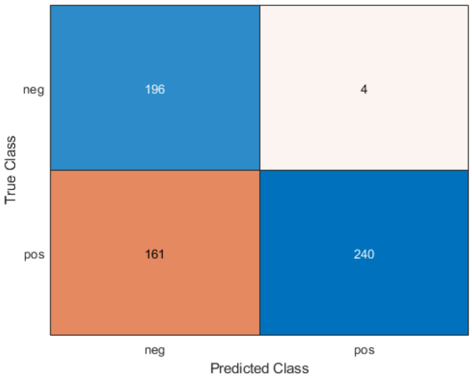


K = 3

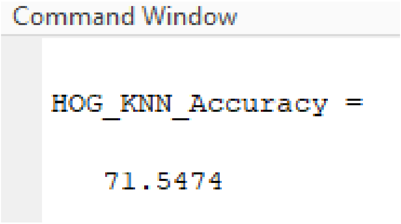


Command Window

```
HOG_KNN_Accuracy =

   72.5458
```

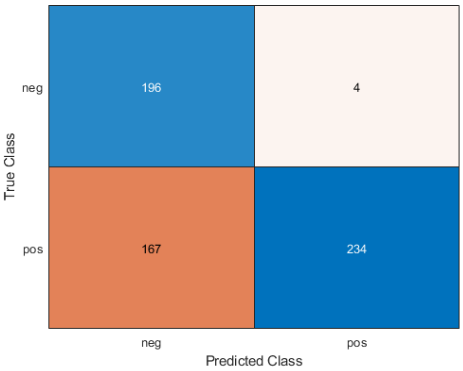


K = 5

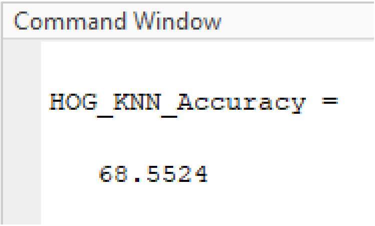


Command Window

```
HOG_KNN_Accuracy =

   71.5474
```

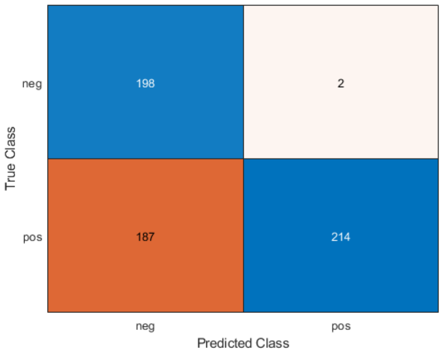


K = 10



Command Window

```
HOG_KNN_Accuracy =

   68.5524
```

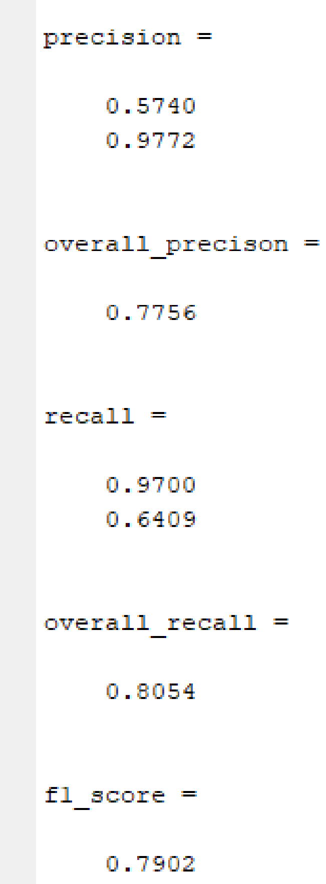


Starting at K = 1, we see that the accuracy starts off reasonably strong at around 75.05% Once we increment by 2 to k= 3, we see a slight decrease to 72.55. Incrementing again by 2 to k = 5, we see the same trend. A decrease in accuracy to 71.55%. Finally, we decided to increment K again to K = 10. With our biggest increase in k, we also saw our largest decrease in accuracy to 68.55% We can gather from this that as K nearest neighbour increases, the accuracy of the HOG KNN model decreases.

When we look at the confusion matrices for each, they tell a similar story. Each iteration of the confusion matrix correctly predicts a negative image 194 to 198 times out of the 200 negative images in the. The accuracy of detecting a negative image increases slightly when K increases. However, as K increases, we see the correctly predicted positive images decrease 257 all the way to 214. K nearest neighbour already wasn't too accurate when K = 1 as it falsely predicted 144 positive images that were in fact negative. This inaccuracy increases to 187 falsely predicted positive images for K = 10.

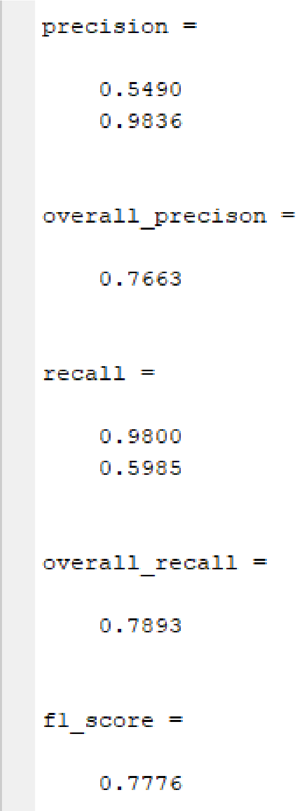
HOG KNN Scores

K=1



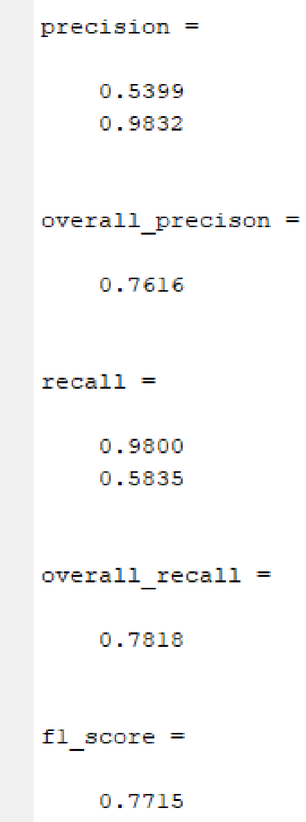
```
precision =

    0.5740
    0.9772


overall_precison =

    0.7756


recall =

    0.9700
    0.6409


overall_recall =

    0.8054


f1_score =

    0.7902
```

K=3



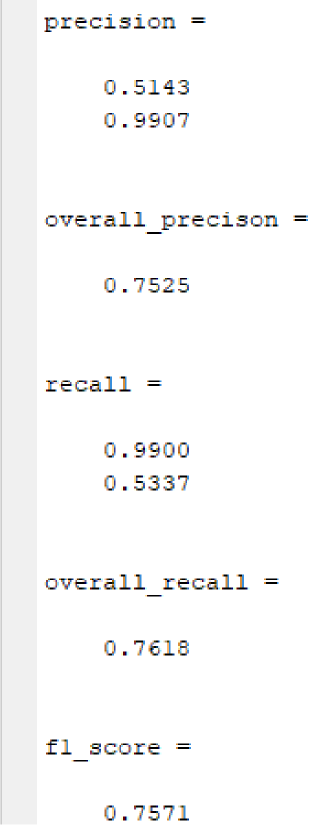
```
precision =

    0.5490
    0.9836


overall_precison =

    0.7663


recall =

    0.9800
    0.5985


overall_recall =

    0.7893


f1_score =

    0.7776
```

K=5



```
precision =

    0.5399
    0.9832


overall_precison =

    0.7616


recall =

    0.9800
    0.5835


overall_recall =

    0.7818


f1_score =

    0.7715
```

K=10



```
precision =

    0.5143
    0.9907


overall_precison =

    0.7525


recall =

    0.9900
    0.5337


overall_recall =

    0.7618


f1_score =

    0.7571
```

Evaluating the KNN values looking at the precision, recall and f1 score, we can see that K=1 only ever so slightly beats out the other 3. We see an overall precision score of 0.7756 for k=1 and it decreases with each iteration of k we tested, all the way to 0.7525 for k=10. While all scores are similar, the k=1 gives us a lower false positive rate compared to the other 3.

For recall we see the same trend between all four values for k. K = 1 gives us an overall recall of 0.8054 and decreases again through each iteration to 0.7618. While all scores are good as they score above 0.5, k = 1 comes out on top because of its higher score.

Finally, we see that F1 scores follow a similar trend. This is to be expected as it considers the recall and precision scores. K =1 gives us a f1 score of 0.7902 and decreases through each iteration of K to 0.7571 for k=10.

With all scores being so similar, the accuracy of the model will not be impacted drastically no matter what value of k we use. But for our feature descriptor, we will be using k=1 for HOG KNN.

Cross Validated HOG SVM Classifier

Our data was divided into a stratified split of 80:20, 80% used for training and 20% used for testing. While these two sets were randomised, the test and accuracy results could depend heavily on how the data was divided to begin with.

To avoid this, we performed cross validation on HOG SVM and KNN classifiers, beginning with SVM. For this we used k-fold cross validation, where the data is randomly divided up into K sets known as folds. One of these folds will be kept as a validation set or unseen set, while the other folds will be used for training. Then the process will be repeated, with a different fold reserved for validation. This will continue until all folds have been used once as the validation set.

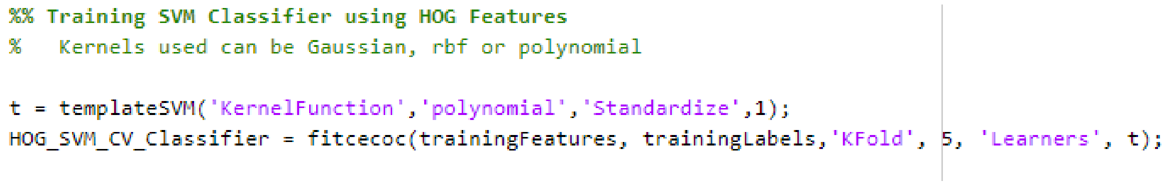
The average loss from all the folds is the k-fold loss. This will reduce the dependency of the loss on how our data was divided. This whole process requires more computing power to complete to fit and evaluate multiple models.

The bigger the value of K, the more time it will take to run but the loss estimate will be more robust.

When deciding on a value for K, we settled on 5. This will divide the 3000 images into 5 sets of 600. One set will be 600, essentially the same as our 80:20 split.

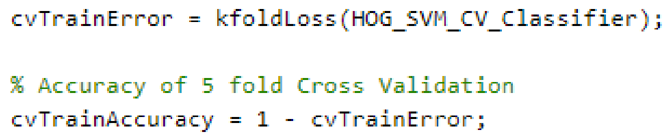
When importing the images into the script, they were not split into testing and training. Instead, the full set of 3000 images was shuffled. The images are then pre-processed into grayscale and the Hog features vectors are extracted with a cell size of [8,8]. These are then stored within ‘trainingFeatures’ and ‘trainingLabels’ are taken from the folder names where the images are stored.

Within our fitcecoc function, we can specify how many folds we wish to use for cross validation. For this we have used a kfold of 5. The ‘templateSVM’ function allows us to state which kernel to use.



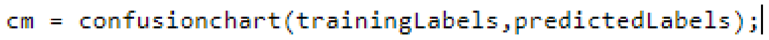
```
%% Training SVM Classifier using HOG Features
%   Kernels used can be Gaussian, rbf or polynomial

t = templateSVM('KernelFunction','polynomial','Standardize',1);
HOG_SVM_CV_Classifier = fitcecoc(trainingFeatures, trainingLabels,'KFold', 5, 'Learners', t);
```

This takes more time compared to the stratified split of previous scripts. Once it has completed running, we can then run the kfoldloss function on the classifier. This will take the average loss from all the folds. From here we can then find accuracy of the cross validated model using ‘cvTrainAccuarcy’ as shown. Both ‘cvTrainError’ and ‘cvTrainAccuarcy’ can both be multiplied by 100 to find out the loss and accuracy in percentages respectively. This was done using the polynomial and gaussian kernels.



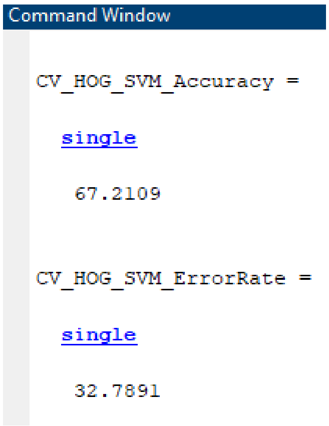
```
cvTrainError = kfoldLoss(HOG_SVM_CV_Classifier);

% Accuracy of 5 fold Cross Validation
cvTrainAccuracy = 1 - cvTrainError;
```

To generate a confusion matrix from our cross validated model, we use the kfoldpredict function on the cross validated SVM model. From here, we can use confusionchart function on out training labels and all predicted labels.

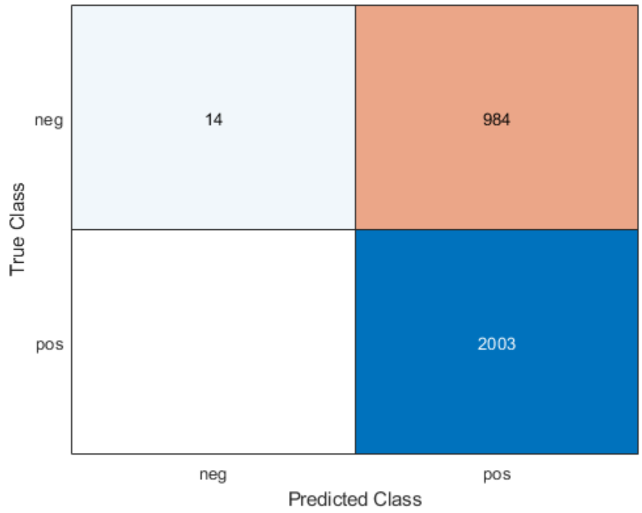


```
cm = confusionchart(trainingLabels,predictedLabels);
```

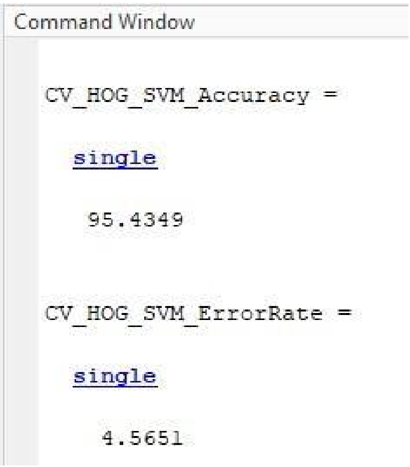
Gaussian Kernel



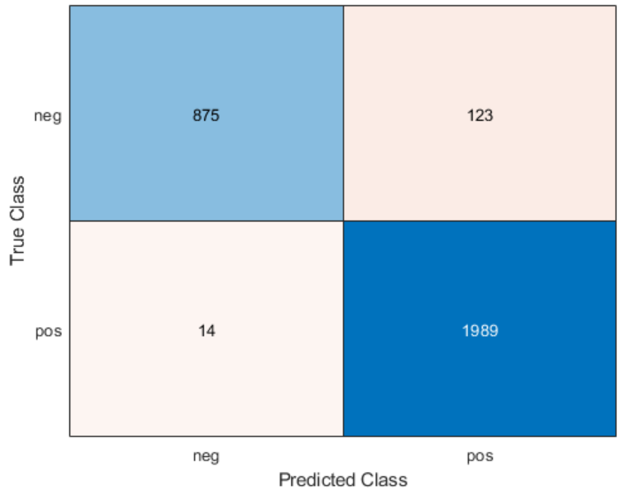
```
Command Window

CV_HOG_SVM_Accuracy =

  single

   67.2109


CV_HOG_SVM_ErrorRate =

  single

   32.7891
```



Polynomial Kernel



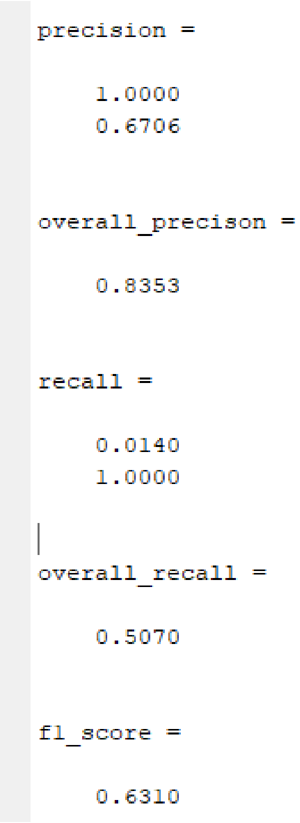
```
Command Window

CV_HOG_SVM_Accuracy =

  single

   95.4349


CV_HOG_SVM_ErrorRate =

  single

    4.5651
```



A quick look at the accuracies for these cross validated SVM, we can see that the cross validated accuracy of the Gaussian kernel takes a small decrease from the stratified accuracy to 67.21% from 67.55%. With the decrease in accuracy being next to nothing, this may be a better model to use in the long run as images have been used as a validation. From the confusion matrix we see that the model only correctly predicted 14 out of 1000 negative images with no people, the other 986 falsely predicted as an image with people. But it correctly predicted 2003 positive images.

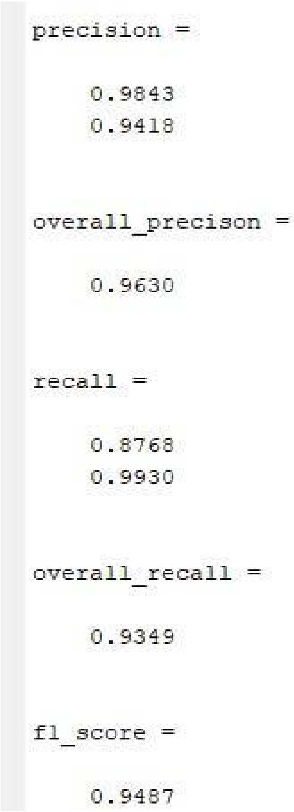
For the polynomial accuracy, we see the same theme; a decrease in cross validated accuracy compared to the stratified accuracy from 97.67% down to 95.43%. At a quick glance it may seem that the stratified model may be better but as stated, with all 3000 images used as a validation set at some point compared to only 600 for stratified and still retaining an almost perfect accurate score, the cross validated model may provide the better results when used in the real world. The confusion matrix shows that the model correctly chooses 875 negative images out of 1000, a far bigger improvement from the Gaussian kernel. For the positive images, the polynomial model got 1989 out of 2003 positive images. While the Gaussian model may have got all positive images correct, the fact that it almost got no negative images shows already that HOG SVM Polynomial is almost a perfect model to use for pedestrian detection.

Gaussian Scores



```
precision =

    1.0000
    0.6706


overall_precison =

    0.8353


recall =

    0.0140
    1.0000


overall_recall =

    0.5070


f1_score =

    0.6310
```

Polynomial Scores



```
precision =

    0.9843
    0.9418


overall_precison =

    0.9630


recall =

    0.8768
    0.9930


overall_recall =

    0.9349


f1_score =

    0.9487
```

Looking at the scores for both the cross validated models, they tell a similar story to the stratified scores. The Gaussian kernel’s scores are good while the scores from the polynomial kernel are far better.

Both precision scores are good with gaussian obtaining a score of 0.8353. While precision for Gaussian almost stayed the same compared to its stratified precision, polynomial saw a bigger decrease from 0.9801 for stratified precision to 0.9630. While it is a bigger decrease, it is still higher than Gaussian and therefore better.

This also applies to the recall, with Gaussian only obtaining slightly above 0.5 at 0.507 compared to Polynomial scoring far greater with a recall of 0.9349. While both are slight decreases from the stratified scores, the fact that polynomial recall is almost twice as much ensures it is a better model with a higher ratio of correctly predicted positive observations.

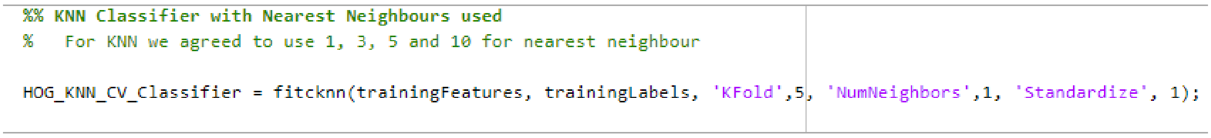
Finally, the F1 score takes the weighted average of both the precision and recall scores. Therefore, it considers the false positives and false negatives, making it more useful compared to the accuracy we calculated. Calculating the f1 score, we see for the Gaussian Kernel the score is 0.6310 while the f1 score for Polynomial is 0.9487. This again follows the same trend, with Gaussian showing little to no difference in its scores compared to its stratified counterpart while Polynomial sees more of a decrease but is still more accurate than the Gaussian Kernel

With these scores calculated, I can see that using the cross validated polynomial kernel for our feature descriptor is the only option as the scores are almost perfect when compared to the Gaussian kernel.

Cross Validated HOG KNN Classifier

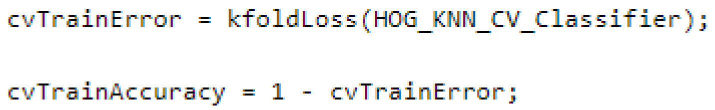
As stated earlier, cross validation was also performed on the KNN classifier with a k fold of 5. When importing the images into the script, they were not split into testing and training. Instead, the full set of 3000 images was shuffled. The images are then pre-processed into grayscale and the Hog features vectors are extracted with a cell size of [8,8]. These are then stored within 'trainingFeatures' and 'trainingLabels' are taken from the folder names where the images are stored.

Within our fitcknn function, we can specify how many folds we wish to use for cross validation. For this we have used a kfold of 5. 'NumNeighbours' was also used with this function to specify how many nearest neighbours to use.



```
%% KNN Classifier with Nearest Neighbours used
%   For KNN we agreed to use 1, 3, 5 and 10 for nearest neighbour

HOG_KNN_CV_Classifier = fitcknn(trainingFeatures, trainingLabels, 'KFold',5, 'NumNeighbors',1, 'Standardize', 1);
```

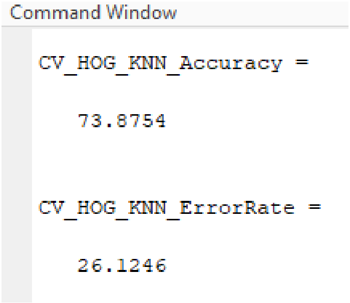
This takes more time compared to the stratified split of previous scripts. Once it has completed running, we can then run the kfoldloss function on the classifier. This will take the average loss from all the folds. From here we can then find accuracy of the cross validated model using 'cvTrainAccuarcy' as shown. Both 'cvTrainError' and 'cvTrainAccuarcy' can both be multiplied by 100 to find out the loss and accuracy in percentages respectively. This was done several times using K Nearest Neighbours of 1,3,5 and 10.



```
cvTrainError = kfoldLoss(HOG_KNN_CV_Classifier);

cvTrainAccuracy = 1 - cvTrainError;
```

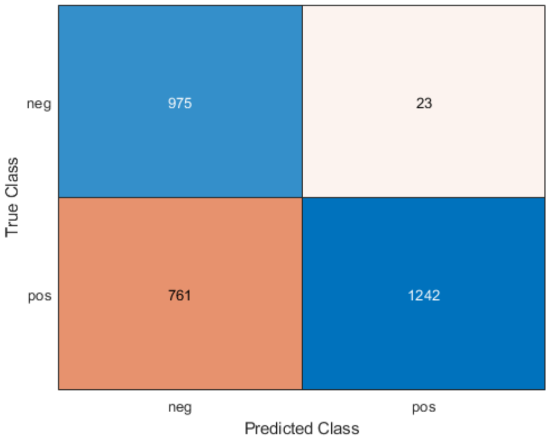
To generate a confusion matrix from our cross validated model, we use the kfoldpredict function on the cross validated SVM model. From here, we can use confusionchart function on out training labels and all predicted labels.

K = 1

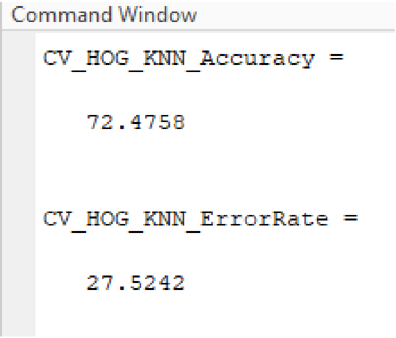


Command Window

```
CV_HOG_KNN_Accuracy =

   73.8754


CV_HOG_KNN_ErrorRate =

   26.1246
```

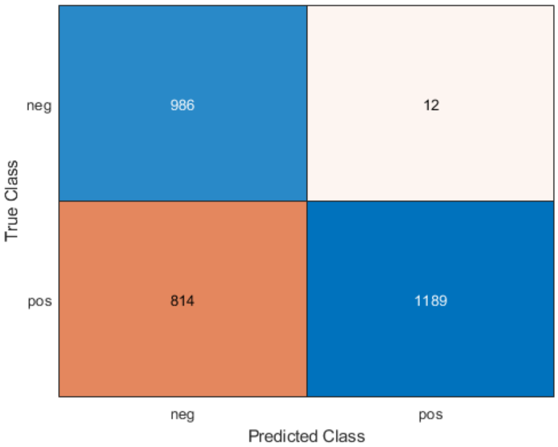


K = 3

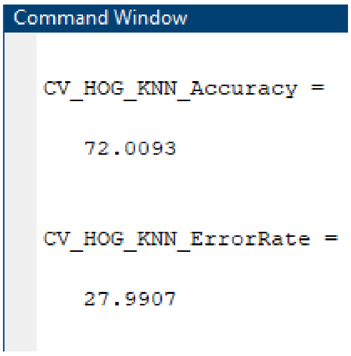


Command Window

```
CV_HOG_KNN_Accuracy =

   72.4758


CV_HOG_KNN_ErrorRate =

   27.5242
```

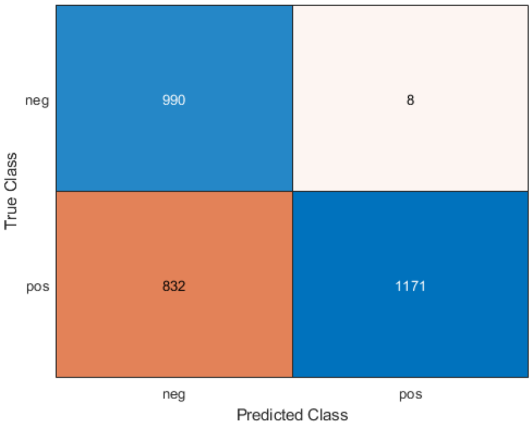


K = 5

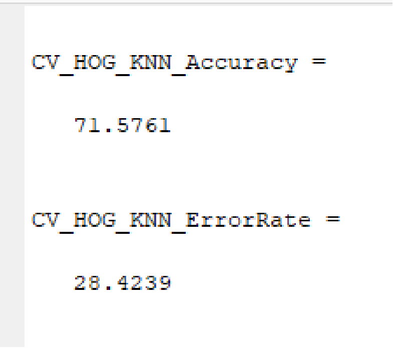


Command Window

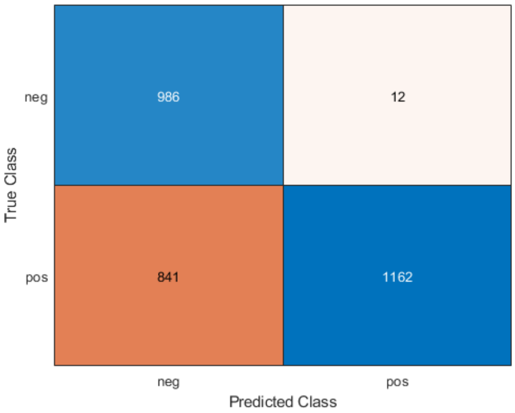
```
CV_HOG_KNN_Accuracy =

   72.0093


CV_HOG_KNN_ErrorRate =

   27.9907
```



K = 10



```
CV_HOG_KNN_Accuracy =

   71.5761


CV_HOG_KNN_ErrorRate =

   28.4239
```

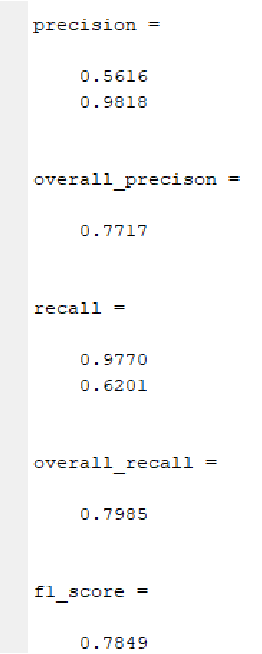


Starting at K = 1, we see that the accuracy starts off reasonably strong at around 73.87% Once we increment by 2 to k= 3, we see a slight decrease to 72.47%. Incrementing again by 2 to k = 5, we see the same trend with accuracy decreasing to 72%. Finally, we incremented K to K = 10. With our biggest increase in k, we also saw our largest decrease in accuracy to 71.57% We can gather from this that as K nearest neighbour increases, the accuracy of the HOG KNN model decreases. But when compared to the stratified accuracies, we see that the accuracy of K = 1 starts off lower, but the decrease in accuracy between each k is lower with K = 10 being around 3% more accurate with cross validation

When we look at the confusion matrices for each, they tell a similar story. Each iteration of the confusion matrix correctly predicts a negative image 975 to 990 times out of the 1000 negative images in the set. The accuracy of detecting a negative image increases slightly when K increases, however from K = 5 and 10 we see that K =5 is more accurate in detecting negative images at 990 compared to 986. For the positive image with people in them, as K increases, we see the correctly predicted positive images decrease 1242 all the way to 1162 images out of 2003. K nearest neighbour already wasn't too accurate when K = 1 as it falsely predicted 761 positive images that were in fact negative. This inaccuracy increases to 841 falsely predicted positive images for K = 10. This follows the same trend as the stratified model earlier, with negative images being correctly chosen but positive images only correctly chosen about 2/3 of the time.

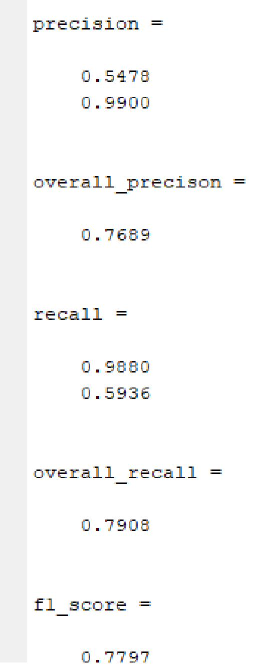
Additional Cross Validated KNN Scores

K=1



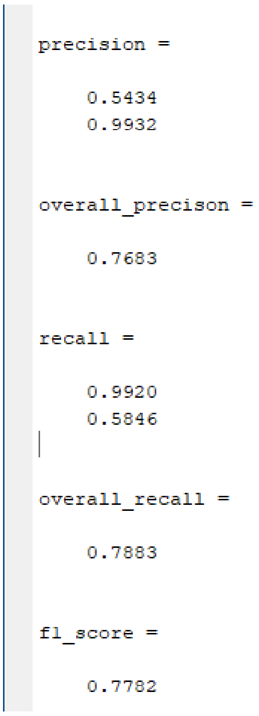
```
precision =

    0.5616
    0.9818


overall_precison =

    0.7717


recall =

    0.9770
    0.6201


overall_recall =

    0.7985


f1_score =

    0.7849
```

K=3



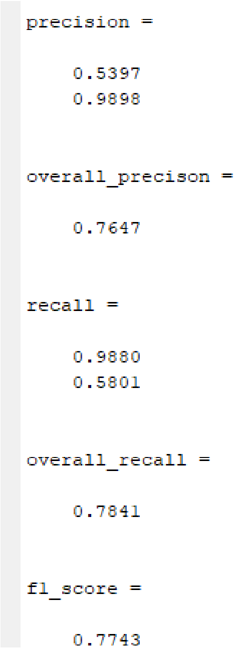
```
precision =

    0.5478
    0.9900


overall_precison =

    0.7689


recall =

    0.9880
    0.5936


overall_recall =

    0.7908


f1_score =

    0.7797
```

K=5



```
precision =

    0.5434
    0.9932


overall_precison =

    0.7683


recall =

    0.9920
    0.5846


overall_recall =

    0.7883


f1_score =

    0.7782
```

K=10



```
precision =

    0.5397
    0.9898


overall_precison =

    0.7647


recall =

    0.9880
    0.5801


overall_recall =

    0.7841


f1_score =

    0.7743
```

Evaluating the KNN values, we can see the values for both the cross validated models, they tell a similar story to the stratified scores.

The KNN values looking at the precision, recall and f1 score, we can see that K=1 only ever so slightly beats out the other 3. We see an overall precision score of 0.7717 for k=1 and it decreases with each iteration of k we tested, all the way to 0.7647 for k=10. All precision values gives us a low false positive rate

For recall we see the same trend between all four values for k. K = 1 gives us an overall recall of 0.7985 and decreases again through each iteration to 0.7841 for K =10. While all scores are good as they score above 0.5, all 4 recall scores are good.

Finally, we see that F1 scores follow a similar trend. This is to be expected as it considers the recall and precision scores. K =1 gives us a f1 score of 0.7849 and decreases through each iteration of K to 0.7743 for k=10. Again, while K = 1 has a higher score, all 4 scores are so similar that any of them can be used for similar results.

With all scores being so similar, the accuracy of the model will not be impacted drastically no matter what value of k we use. But for our feature descriptor, we will be using k=1 for HOG KNN Cross Validation.

Feature Descriptor - Full Image

Full Image KNN Classifier

The first classifier we applied to our full image feature descriptor was the k-nearest neighbour classifier. We chose this as it performs reasonably well over large datasets, which fits well with our 3000 training images.

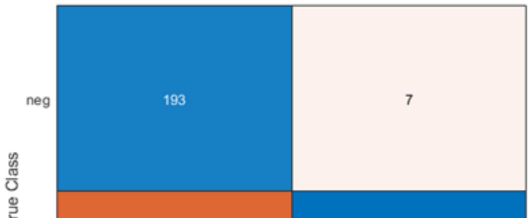
As with our HOG feature descriptor, we also decided on an 80/20 split for our training images. The 80% were again randomised across the positive and negative images in order to avoid bias when selecting positive or negative images to train the classifier. Their labels were also identified, assigned and used for training the classifier. The remaining 20% were then used to test the classifier once trained. The parameters and logic used for the splitting of the training images was the same as used for HOG.

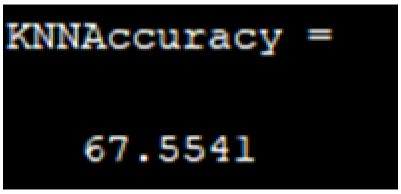
Each image in the training set was then underwent pre-processing by being converted into grayscale, reshaped into a single vector and each value in the vector was converted to a double before being passed to the KNN classifier.

As previously highlighted, we used values of 1, 3, 5 and 10 for our K number of neighbours. This wide range helped us to remove the possibility of noise and outliers having an impact on our training results.

When we tested our KNN model for the different values of K, these are the accuracy results we achieved:

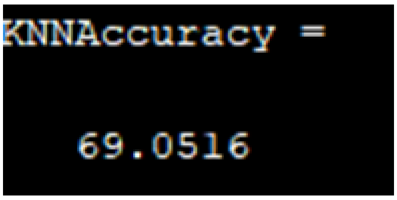
K = 1



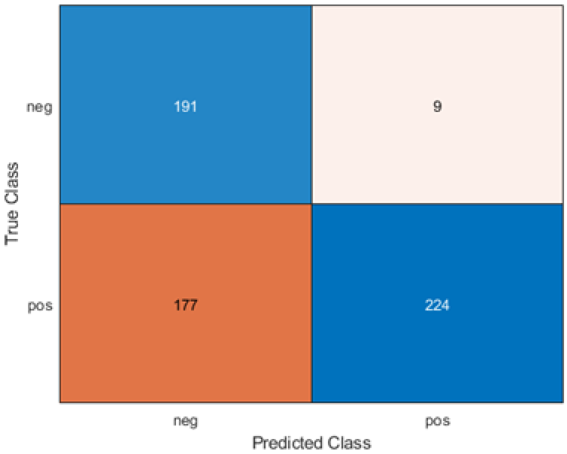


```
KNNAccuracy =

   67.5541
```

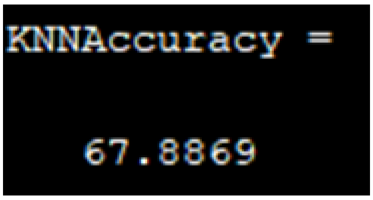
K = 3



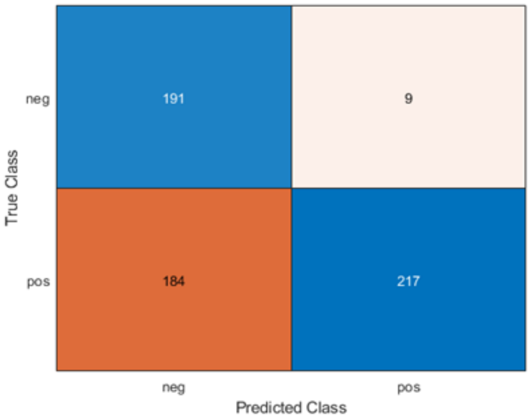
```
KNNAccuracy =

   69.0516
```



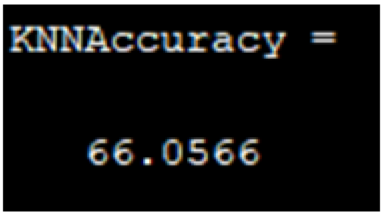
K = 5



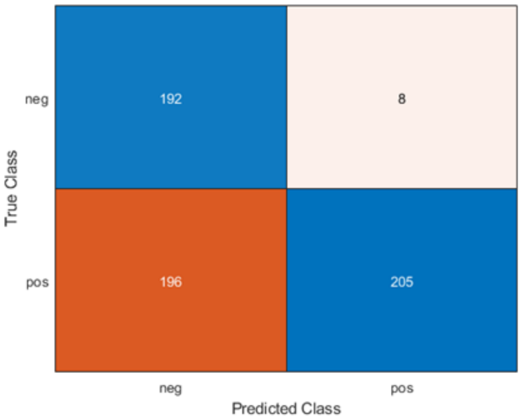
```
KNNAccuracy =

   67.8869
```



K = 10



```
KNNAccuracy =

   66.0566
```

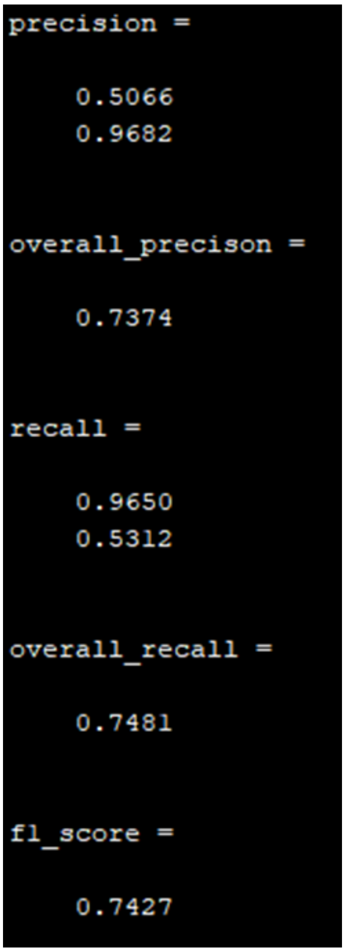


For K = 1 the accuracy was 67.55%, whic K = 1 only 1 neighbour is considered so the lower accuracy may mean that outliers in the training have led to incorrect classifications. For K= 5 the accuracy was 67.87% and 66.06% for K = 10. With a gradual decrease in accuracy from K = 3 to K = 10 by almost 3% we can conclude that training this classifier with values of greater than K = 3 will lead to decreasing accuracies.

This is further reflected when looking at the confusion matrices for each value of K. For values of greater than K = 3 the number of false positive detections of pedestrians consistently increases while the number true negative detections of pedestrians decreases.

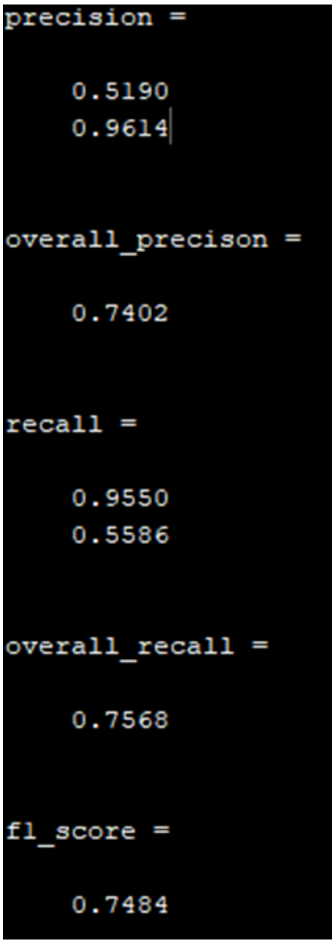
Additional KNN scores

K = 1



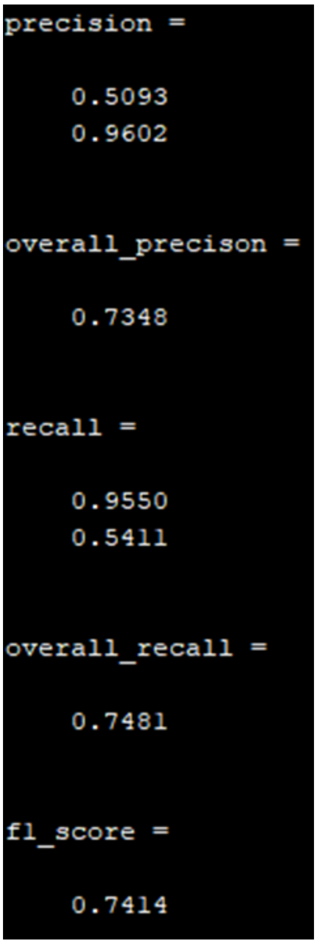
```
precision =

    0.5066
    0.9682


overall_precison =

    0.7374


recall =

    0.9650
    0.5312


overall_recall =

    0.7481


f1_score =

    0.7427
```

K = 3



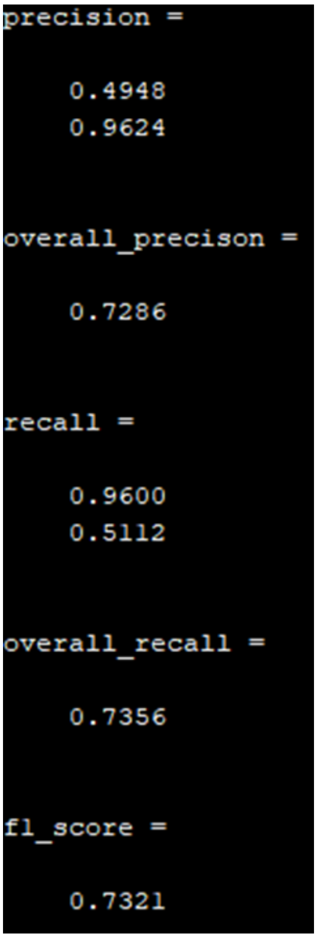
```
precision =

    0.5190
    0.9614


overall_precison =

    0.7402


recall =

    0.9550
    0.5586


overall_recall =

    0.7568


f1_score =

    0.7484
```

K = 5



```
precision =

    0.5093
    0.9602


overall_precison =

    0.7348


recall =

    0.9550
    0.5411


overall_recall =

    0.7481


f1_score =

    0.7414
```

K = 10



```
precision =

    0.4948
    0.9624


overall_precison =

    0.7286


recall =

    0.9600
    0.5112


overall_recall =

    0.7356


f1_score =

    0.7321
```

When comparing the overall precision and overall recall for each value of K, a value of K = 3 is again the most ideal of the values. The trend here is similar to the accuracy values with precision and recall rates increasing up to K = 3 and then decreasing for larger values. The suitability of each value of K is further shown in the F1 scores. The score increases by 0.0057 to reach 0.7484 from K = 1 to 3 and then continues to decrease over K = 5 and has fallen by 0.0163 to 0.7321 for K = 10. This is to be expected as it takes into account the precision and recall scores which also show the same trend. Therefore K =3 is the most suitable value for our full image KNN classifier.

Cross Validated Full Image KNN Classifier

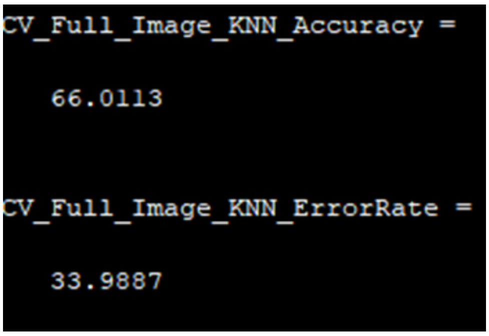
As we used a randomised 80/20 splitting the training images into training and testing sets, this meant that the quality of the results was based partly on how the images were randomly divided. To try and alleviate this we re-ran the full image KNN training and testing having applied k-fold cross validation.

As previously stated we chose a k-fold value of K = 5 so that our 3000 training images were divided into 5 sets. We chose a value of 5 to mirror our earlier 80/20 split of the images before applying cross validation so that we could compare the results directly.

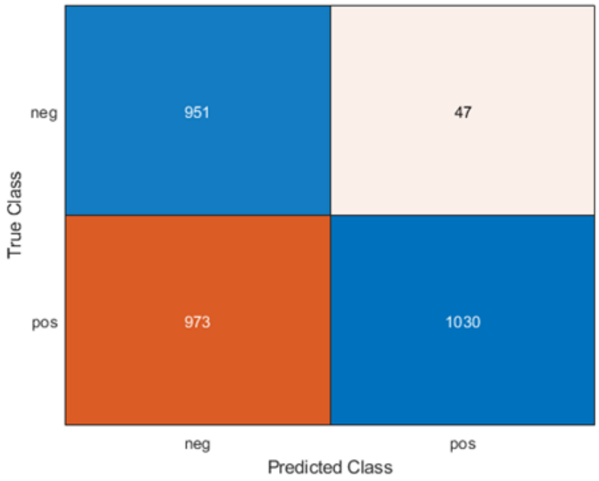
The training and testing process was essentially the same with four of the folds (80% of images) being used to train the classifier and the remaining one fold (20% of images) being used to test the trained model.

The accuracy results of applying this k-fold cross validation were as follows:

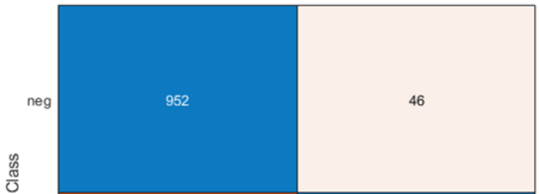
K = 1

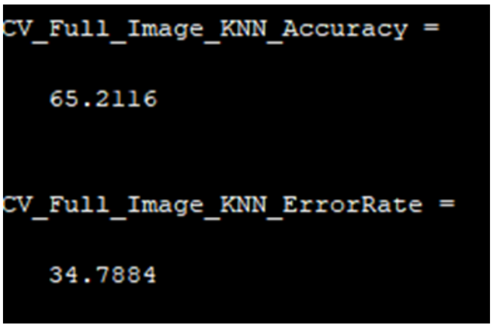


```
CV_Full_Image_KNN_Accuracy =

   66.0113


CV_Full_Image_KNN_ErrorRate =

   33.9887
```



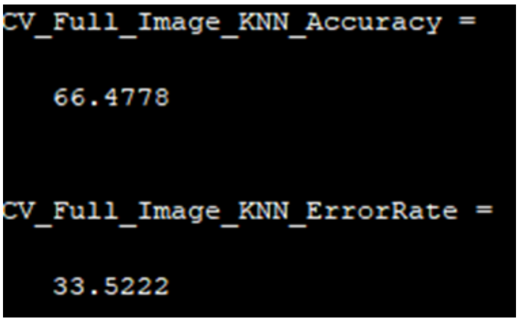
K = 3



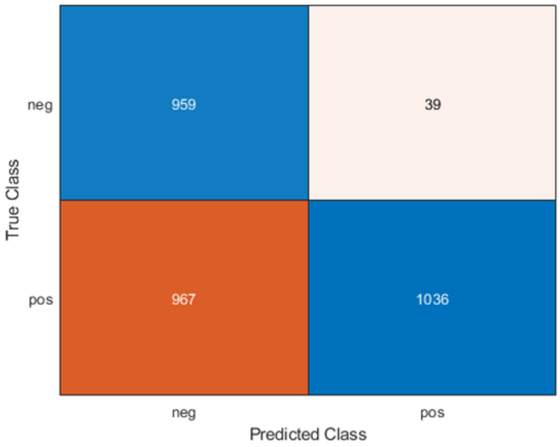


```
CV_Full_Image_KNN_Accuracy =

   65.2116


CV_Full_Image_KNN_ErrorRate =

   34.7884
```

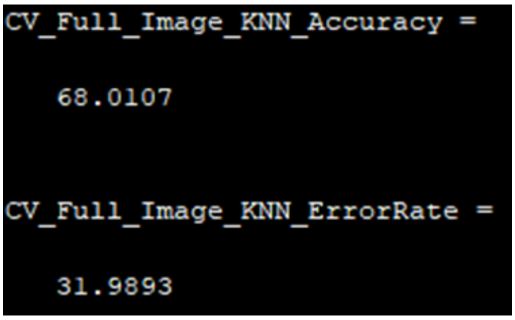
K = 5



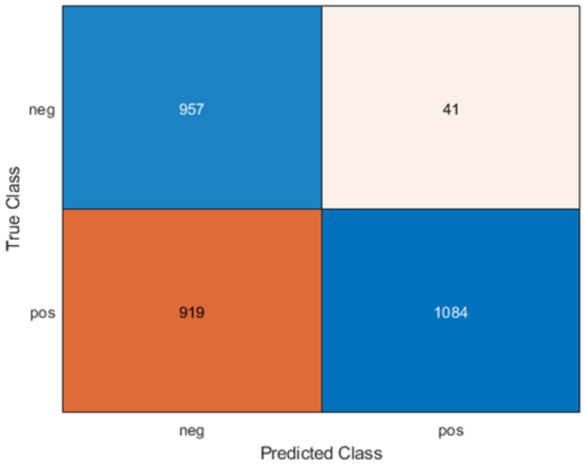
```
CV_Full_Image_KNN_Accuracy =

   66.4778


CV_Full_Image_KNN_ErrorRate =

   33.5222
```



K = 10



```
CV_Full_Image_KNN_Accuracy =

   68.0107


CV_Full_Image_KNN_ErrorRate =

   31.9893
```

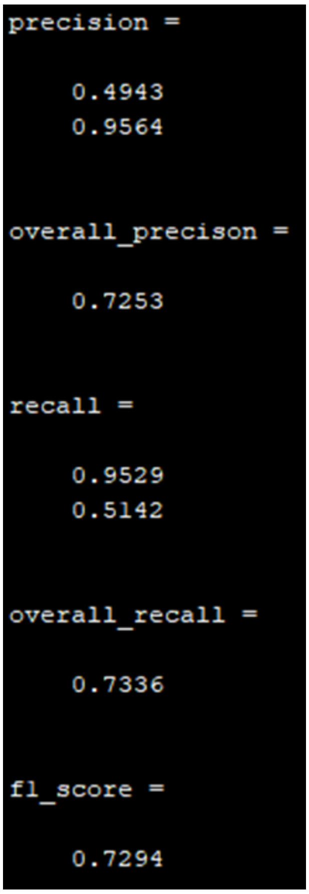


Having applied k-fold cross validation to divide our images into training and testing sets, our classifier has produced quite different results than the previous random

80/20 split. This time K = 1 has achieved an accuracy of 0.80% higher than K = 3 and also has a lower false positive rate. For K = 5 the accuracy is higher than K = 3 and is 0.47% greater than K = 1. Further conflicting with our random 80/20 split results, K = 10 produced the highest accuracy at 68.01% and lowest false positive rate being 48 lower than the second lowest.

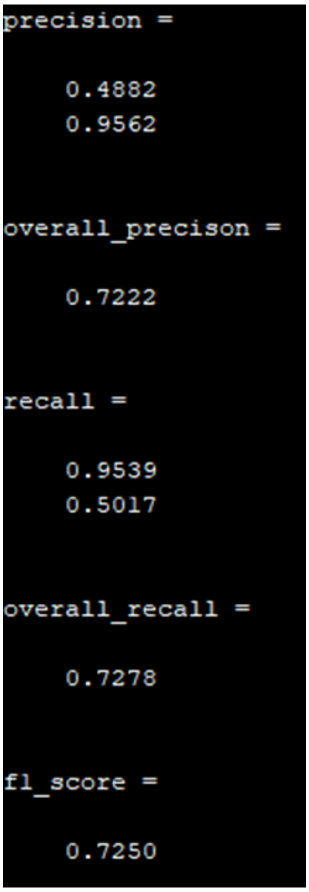
Additional Cross Validated KNN scores

K = 1



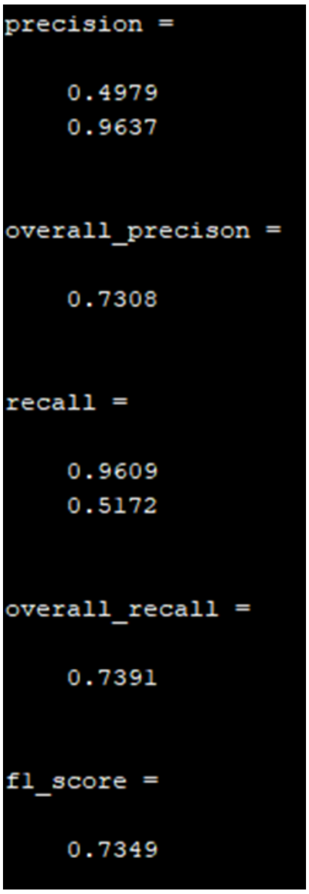
```
precision =

    0.4943
    0.9564


overall_precison =

    0.7253


recall =

    0.9529
    0.5142


overall_recall =

    0.7336


f1_score =

    0.7294
```

K = 3



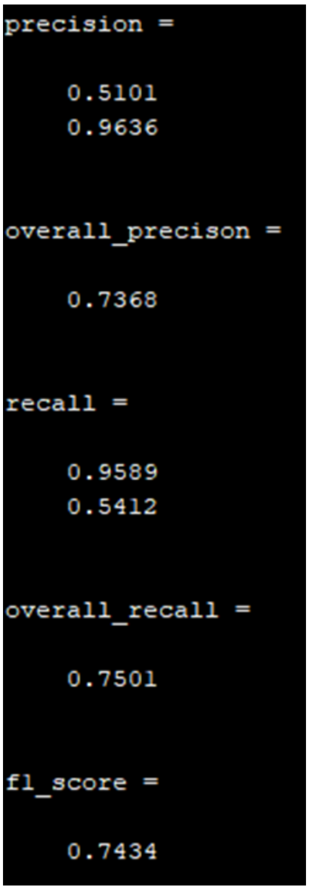
```
precision =

    0.4882
    0.9562


overall_precison =

    0.7222


recall =

    0.9539
    0.5017


overall_recall =

    0.7278


f1_score =

    0.7250
```

K = 5



```
precision =

    0.4979
    0.9637


overall_precison =

    0.7308


recall =

    0.9609
    0.5172


overall_recall =

    0.7391


f1_score =

    0.7349
```

K = 10



```
precision =

    0.5101
    0.9636


overall_precison =

    0.7368


recall =

    0.9589
    0.5412


overall_recall =

    0.7501


f1_score =

    0.7434
```

Looking at the overall precision and overall recall rates for each value of K, we can see the same trend from the accuracy values. The overall precision and recall starts at 0.7253 and 0.7336 for K = 1 which then both decrease slightly for K = 3. From K = 3 to K = 5 there is a large increase of 0.0086 in overall precision and 0.0113 in overall recall. This trend continues for K = 10 as the overall precision increases again from K = 5 by 0.0060 to 0.7368 and overall recall increases by 0.11 to 0.7501. This is clearly shown in the F1 score as it drops from K = 1 to 3 and then continues to increase up to 0.7434 for K = 10. Therefore, the best value for this trained KNN classifier is K = 10.

Due to the difference in results for the two trained KNN classifiers, we can conclude that how the images are randomly split between the training and testing sets can have a noticeable impact on the accuracy and precision of the classifier.

For our first KNN model using K = 3 we achieved a highest accuracy of 69.05% and F1 score of 0.7484 and our second model, using K = 10, achieved a highest accuracy of 68.01% and F1 score of 0.7434. Although the difference is not significant, as the first trained model has a 1.04% higher accuracy and 0.005 higher F1 score, we conclude that the best value to use for our KNN classifier is K = 3.

Full Image SVM Classifier

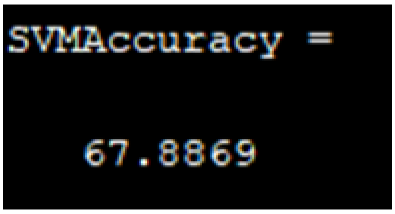
The second classifier we applied to our full image feature descriptor was the SVM classifier. We chose this as it uses a flexible kernel-based system making it a powerful classifier and also has a fast testing stage. This makes it suitable to train our pedestrians due to the wide variety of positive and negative images we used to train it.

We again applied the 80/20 split over our 3000 training images to create training and testing image sets. The pre-processing each image underwent was the same as for the KNN classifier and these images were then passed into the SVM classifier.

We decided on using two kernels: Gaussian and polynomial. The Gaussian kernel was chosen as it is quite a powerful kernel to apply and would likely give us a high accuracy. We chose the polynomial kernel to compare it to as it would provide a different boundary between our binary training images in the model it produced and is also more time effective than Gaussian.

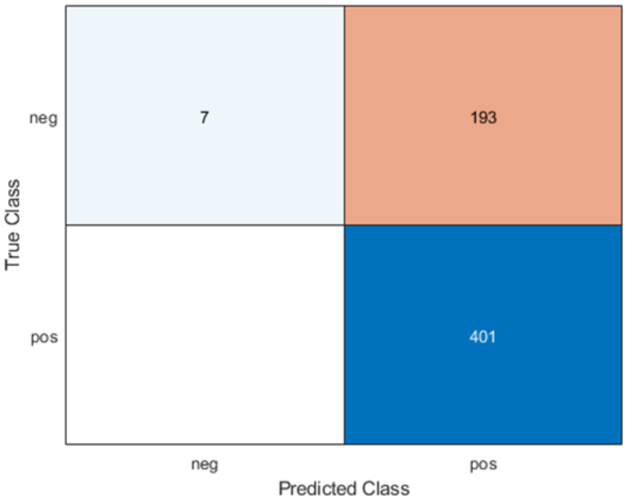
When our trained SVM models were tested these are the results that were achieved:

Gaussian Kernel

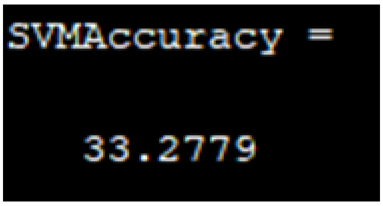


```
SVMAccuracy =

   67.8869
```

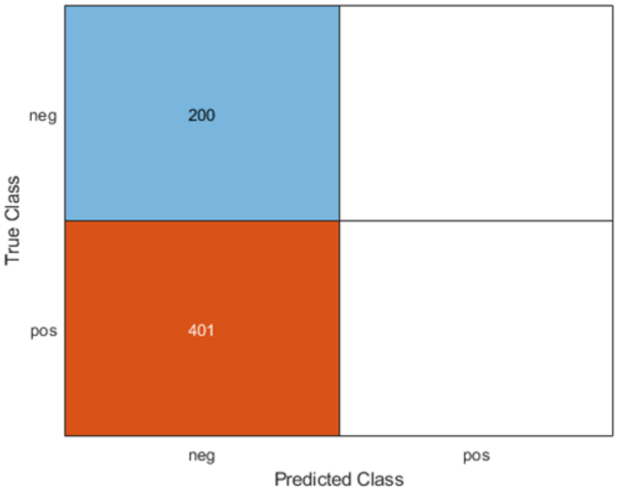
-



Polynomial Kernel

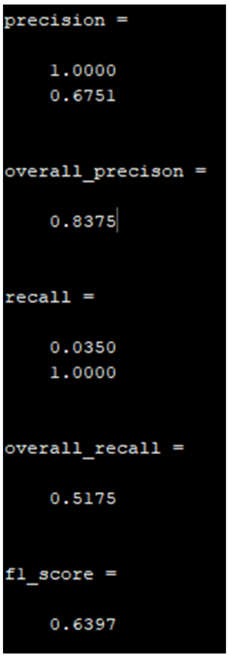


```
SVMAccuracy =

   33.2779
```

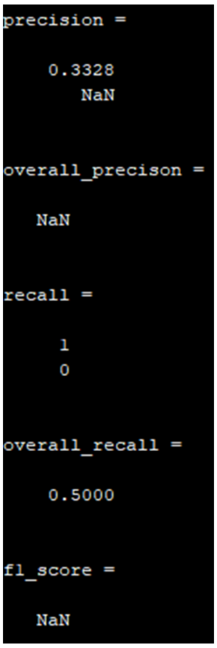


The Gaussian kernel achieved a 67.89% accuracy which was 34.61% higher than the polynomial kernel on the same training and testing sets. Looking at the confusion matrices for the two kernels helps to show us why these accuracies were produced. The Gaussian kernel correctly identified 401 out of 600 testing images but incorrectly identified 193 negative images as having pedestrians. This shows that although it correctly identified a large amount of images, it was prone to false positives. This explains the high accuracy although the model was only able to detect 7 negative images over the test set. The polynomial kernel's confusion matrix tells the opposite story, being able to detect some of the negative images but none of the positive images. No images were identified as having pedestrians and 401 were incorrectly identified as not having pedestrians, with only 200 being correctly identified as not having pedestrians. Based on these values, the Gaussian kernel is the clear winner as the polynomial kernel was unable to detect positive pedestrian images.

Gaussian Scores   Polynomial Scores



```
precision =

    1.0000
    0.6751


overall_precison =

    0.8375


recall =

    0.0350
    1.0000


overall_recall =

    0.5175


f1_score =

    0.6397
```



```
precision =

    0.3328
       NaN


overall_precison =

    NaN


recall =

     1
     0


overall_recall =

    0.5000


f1_score =

    NaN
```

Looking at the additional scores for the two kernels we reach the same conclusions. The Gaussian kernel had an overall precision of 0.8375 which is quite a good score but the overall recall rate was only 0.5175. The overall precision was high because of the seven identified true negative images, all of them were correctly identified. This led to the model producing 100% precision for detecting negative images although only seven images were detected. The recall rate here is a much better metric to evaluate as it shows that only 0.5175 images overall were identified as having the correct labels, meaning the test images were correctly identified only 51%

of the time. For our polynomial kernel, our overall precision was NaN. This was because no positive images were identified and therefore an average of the positive and negative identification precisions could not be calculated. The precision of identifying a negative image was only 0.3328, meaning that 0.6672 of the time the image was incorrectly identified. The recall rate for polynomial was 0.5 and this is only 0.0175 less than the Gaussian kernel. Although having only identified negative images and no positive images the recall rate for polynomial is therefore not a useful metric. As we could not calculate an overall precision for the polynomial kernel, its F1 score could also not be calculated as it takes into account the overall precision and overall recall scores. Therefore, we cannot directly compare the F1 scores of each kernel despite Gaussian producing a score of 0.6397.

As the polynomial kernel was unable to identify any positive pedestrian images and only had a 0.3328 precision for its negative images, the Gaussian kernel is the winning kernel. Although it was only able to identify 7 negative images, its 0.6751 precision for identifying positive pedestrian images makes it the best of the two kernels to set this SVM classifier's parameters.

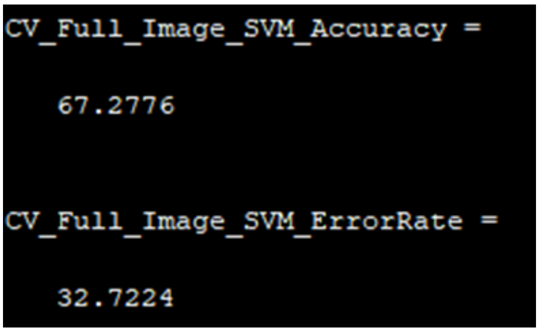
Cross Validated Full Image SVM Classifier

As with the KNN classifier, the random 80/20 split of the training images into training and testing sets may have had an impact on the model and testing results produced by each kernel. Therefore, we trained and tested another model with k-fold cross validation applied. We also chose a k-fold value of 5 to mirror the 80/20 split from the previous full image SVM model so that we could directly compare the results.

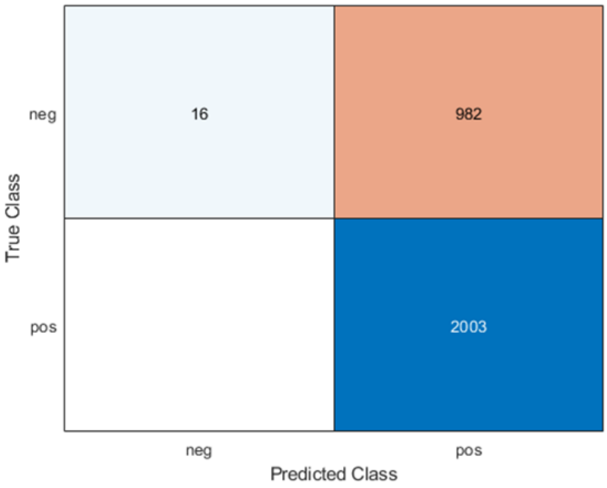
The setup was the same as previously, with Gaussian and polynomial kernels being chosen as the kernel parameters for our SVM classifier.

The accuracy and confusion matrix results were as follows:

Gaussian Kernel

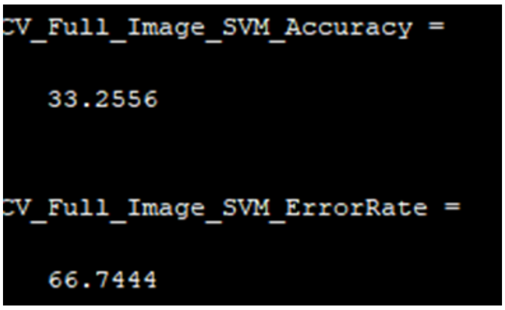


```
CV_Full_Image_SVM_Accuracy =

   67.2776


CV_Full_Image_SVM_ErrorRate =

   32.7224
```

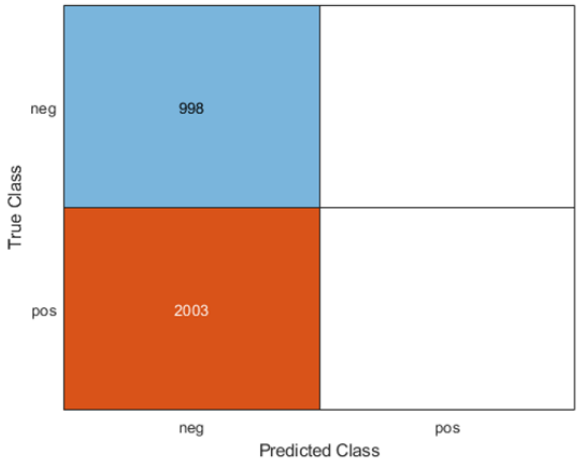


True Class \ Predicted Class	neg	pos
neg	16	982
pos		2003

Polynomial Kernel

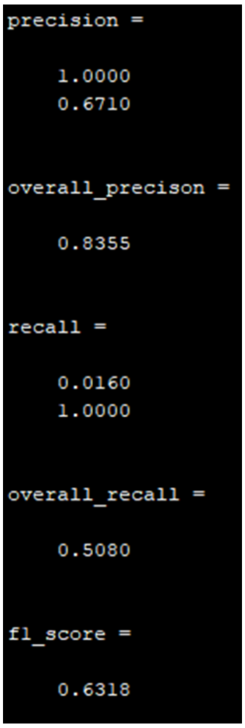


```
CV_Full_Image_SVM_Accuracy =

   33.2556


CV_Full_Image_SVM_ErrorRate =

   66.7444
```

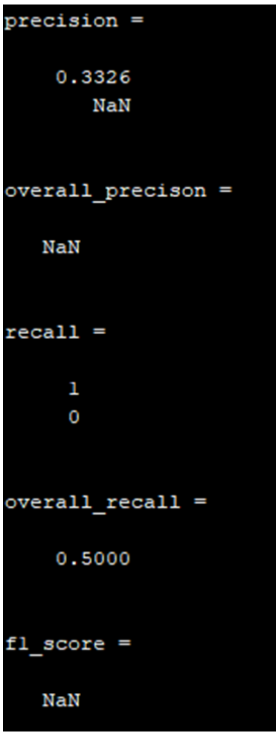


The results here are very close to the previous SVM results for both kernels. Here, Gaussian had an accuracy of 67.23% with an error rate of 32.72%. It also had 2003 true positives, 982 false positives and 16 true negative results. The polynomial had 33.26% accuracy with an error rate of 66.74%. Its confusion matrix shows there were 998 true negative, 2003 false negatives and no positive detections at all. Again, the two kernels are at opposites with Gaussian being able to mostly only produce positive identifications and polynomial only being able to produce negative identifications.

Gaussian Scores **Polynomial Scores**



```
precision =

    1.0000
    0.6710


overall_precison =

    0.8355


recall =

    0.0160
    1.0000


overall_recall =

    0.5080


f1_score =

    0.6318
```



```
precision =

    0.3326
       NaN


overall_precison =

   NaN


recall =

     1
     0


overall_recall =

    0.5000


f1_score =

   NaN
```

The additional scores for this trained model are also very similar to the previous model. Gaussian gave an overall precision of 0.8355, but this is not an ideal metric to use as the 100% correct negative identifications was across only 16 images. The positive precision metric was 0.6710 which can be used as there were 2,985 total true and false positive identifications. The overall recall rate shows only 0.5080 images were identified correctly, which is clearly shown when looking at the Gaussian confusion matrix. Again, the polynomial kernel gave an overall precision of NaN due to no positive images being identified. With a recall rate of 0.5, only 50% of the negative detections by the polynomial kernel were correct. The Gaussian kernel produced a F1 score of

0.6318 but as the polynomial kernel produced a F1 score of NaN these are again not directly comparable.

With the polynomial kernel being unable to correctly identify any positive pedestrian images, when using both the 80/20 and k-fold cross validation, it is not a suitable kernel for our SVM classifier. For both models the Gaussian kernel produced very similar results. We compare the F1 score of them both as it takes into account the precision and recall scores. The first Gaussian model had an F1 score of 0.6397 and the second had a score of 0.6318. Therefore we can conclude that the first model using the 80/20 split in training images has produced the best model for full image SVM classifier. As the Gaussian F1 score when using k-fold cross validation was very close to the 80/20 split Gaussian F1 score, we can also conclude that for SVM, the randomness of the image split is not a significant factor when training the model.

Feature Descriptor - Dimensionality Reduction

Principal Component Analysis (PCA) is the most popular and widely used dimensionality reduction technique. PCA is a versatile tool which allows the dimensions of a data set to be reduced to increase its interpretability and make it easier to visualise while minimising the loss of information. Consequently, by reducing the number of dimensions, it also reduces the time complexity. This makes PCA suitable as a feature descriptor itself as well as being used in conjunction with other feature descriptors to reduce their time complexity.

For the classifier, our group tested both the KNN and SVM classifiers as they are both popular and widely used. Testing with both classifiers is also good as they are polar opposites when it comes to the dealing with the size of the datasets and number of dimensions. For example, SVM is better for smaller datasets and can produce accurate results in higher dimensions. On the other hand, KNN struggles with higher dimensions and generally works well with large datasets albeit at the cost of speed and efficiency.

For testing the classifiers, we used a training and testing split of 80:20 as well as using cross validation with a k-fold of 5. As mentioned before, this splits the data set into 5 sets or folds, and one set is chosen as the testing set and the rest as the training set. This is repeated until each set has the chance to be chosen as the testing set.This procedure allows us to obtain an unbiased estimate of the skill of our machine learning models.

KNN Classifier

For the KNN classifier, it is not easy to determine the optimal number of neighbours so as a group we decided to use K = 1, 3, 5, and 10.

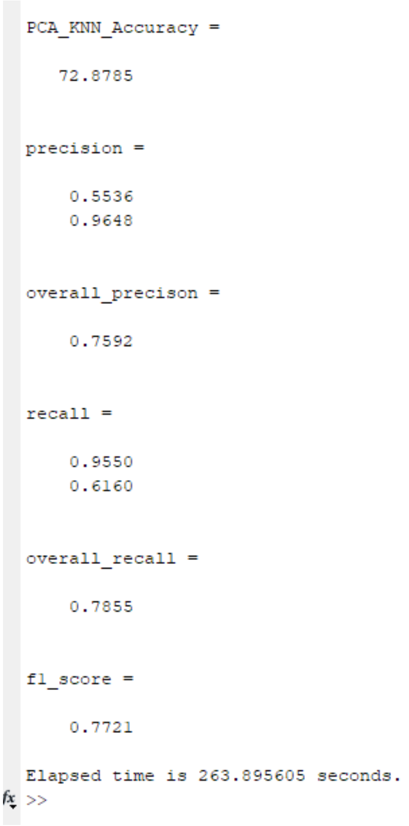
For the 80:20 split, K = 1 gave us the highest accuracy which is 72.88%. Looking at other values of K, there is a trend of decreasing accuracy as the value of K increases.

For the k-fold cross validation, the opposite seems to be true, the highest accuracy achieved was 70.61% and looking at the values of K, the higher K is the more accurate it is.

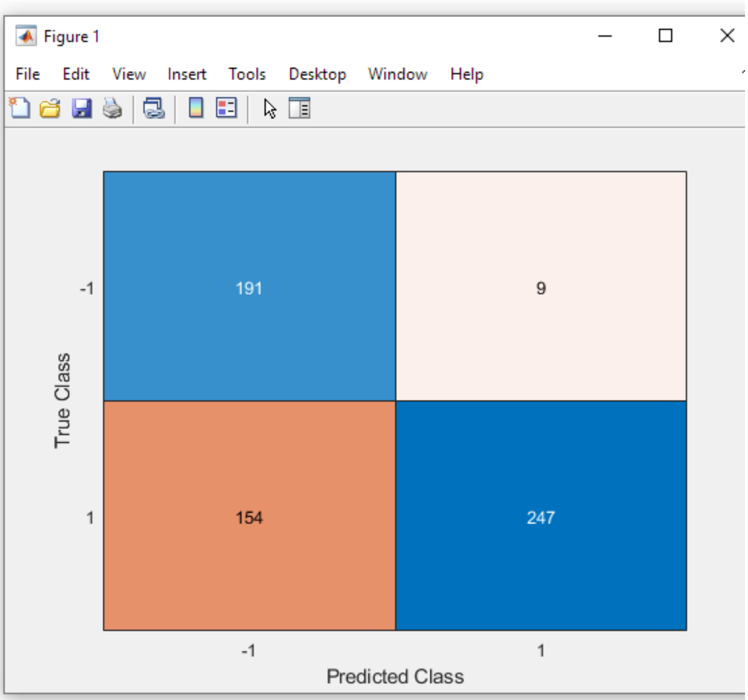
Overall, for KNN, the highest accuracy achieved is 72.88% which is also the highest between the two classifiers using PCA. KNN is the most suitable combination with PCA due to its high accuracy and F1 score but also due to it being a simple classifier. Due to KNN struggling with higher dimensions it is appropriate to pair it with PCA which can reduce the dimensions.

PCA KNN Accuracies and Scores

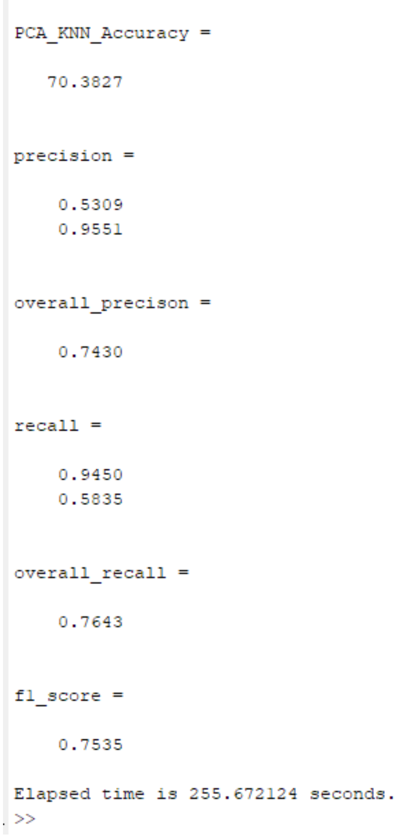
K = 1



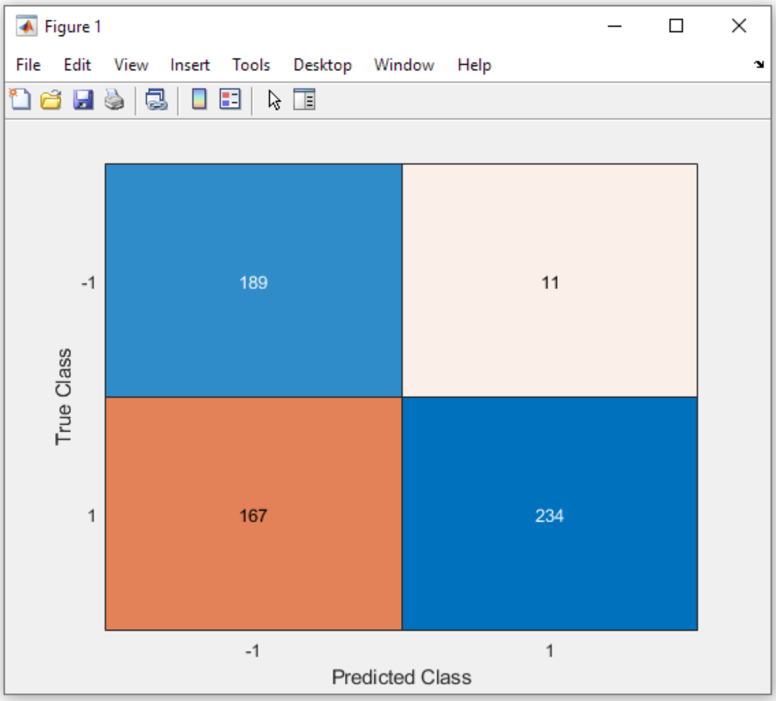
```
PCA_KNN_Accuracy =

   72.8785


precision =

    0.5536
    0.9648


overall_precison =

    0.7592


recall =

    0.9550
    0.6160


overall_recall =

    0.7855


f1_score =

    0.7721

Elapsed time is 263.895605 seconds.
>>
```



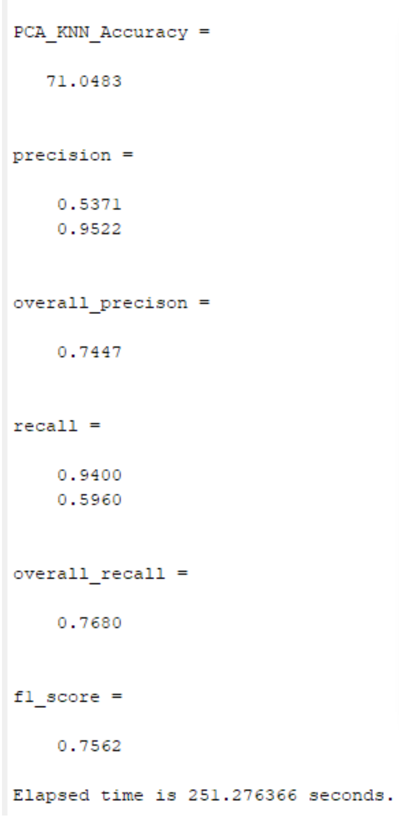
K = 3



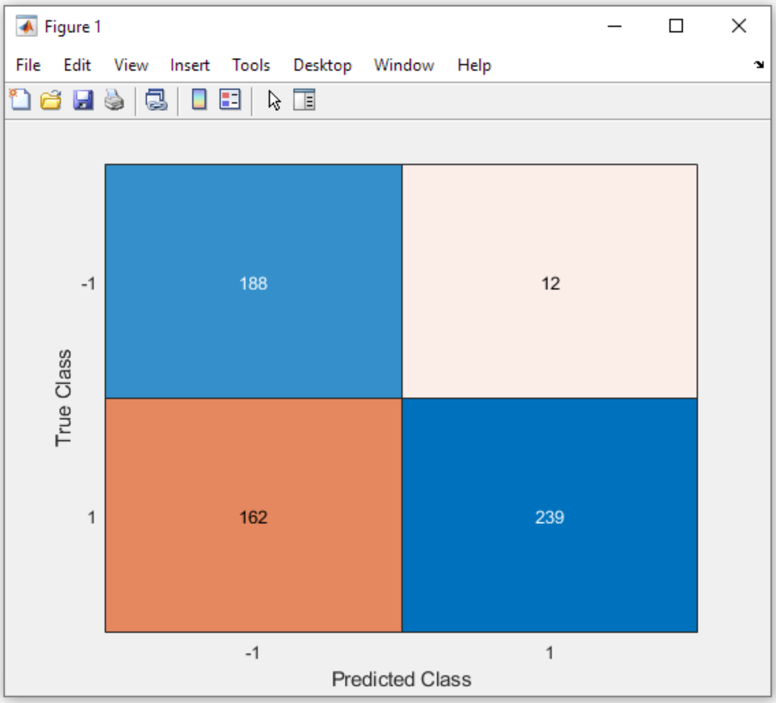
```
PCA_KNN_Accuracy =

   70.3827


precision =

    0.5309
    0.9551


overall_precison =

    0.7430


recall =

    0.9450
    0.5835


overall_recall =

    0.7643


f1_score =

    0.7535

Elapsed time is 255.672124 seconds.
>>
```



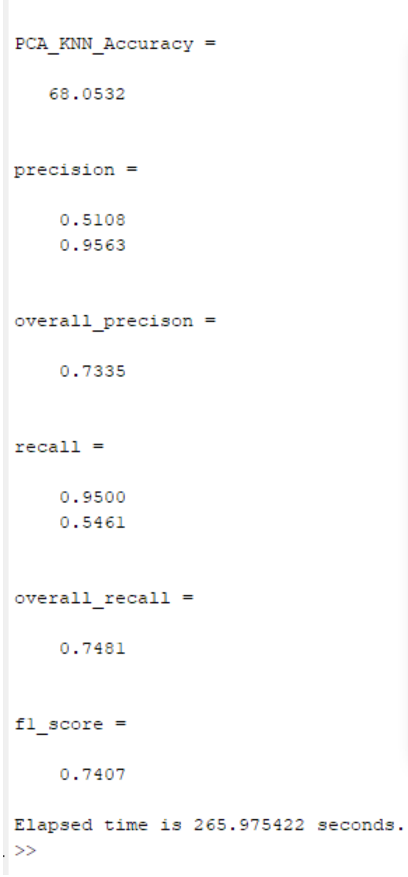
K = 5



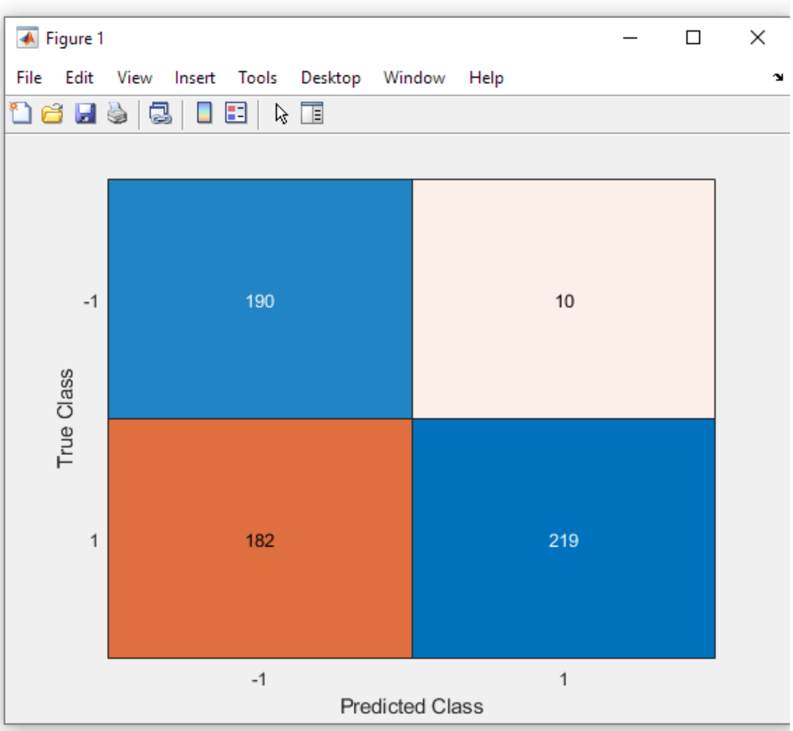
```
PCA_KNN_Accuracy =

   71.0483


precision =

    0.5371
    0.9522


overall_precison =

    0.7447


recall =

    0.9400
    0.5960


overall_recall =

    0.7680


f1_score =

    0.7562

Elapsed time is 251.276366 seconds.
```



K = 10



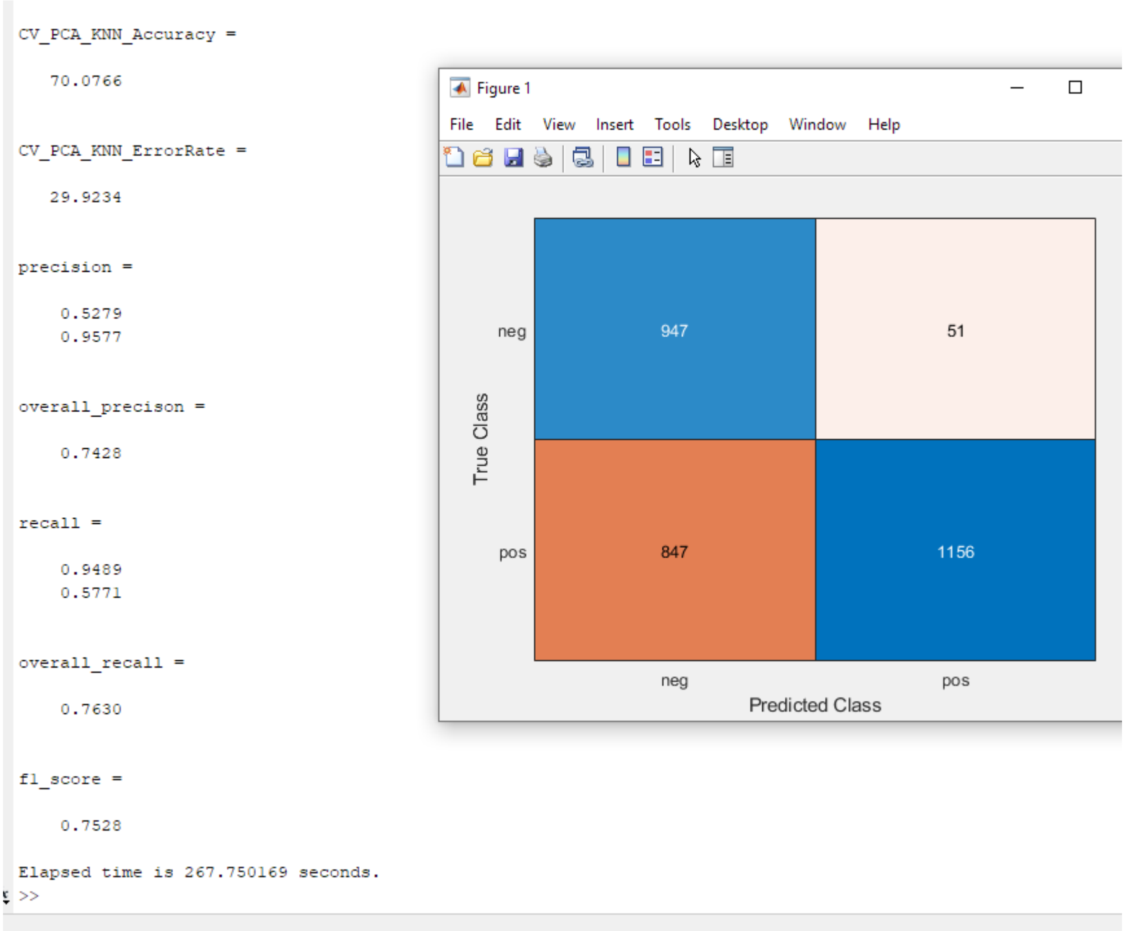
```
PCA_KNN_Accuracy =

   68.0532


precision =

    0.5108
    0.9563


overall_precison =

    0.7335


recall =

    0.9500
    0.5461


overall_recall =

    0.7481


f1_score =

    0.7407

Elapsed time is 265.975422 seconds.
>>
```



True Class \ Predicted Class	-1	1
-1	190	10
1	182	219

Cross Validated PCA KNN Accuracies and Scores

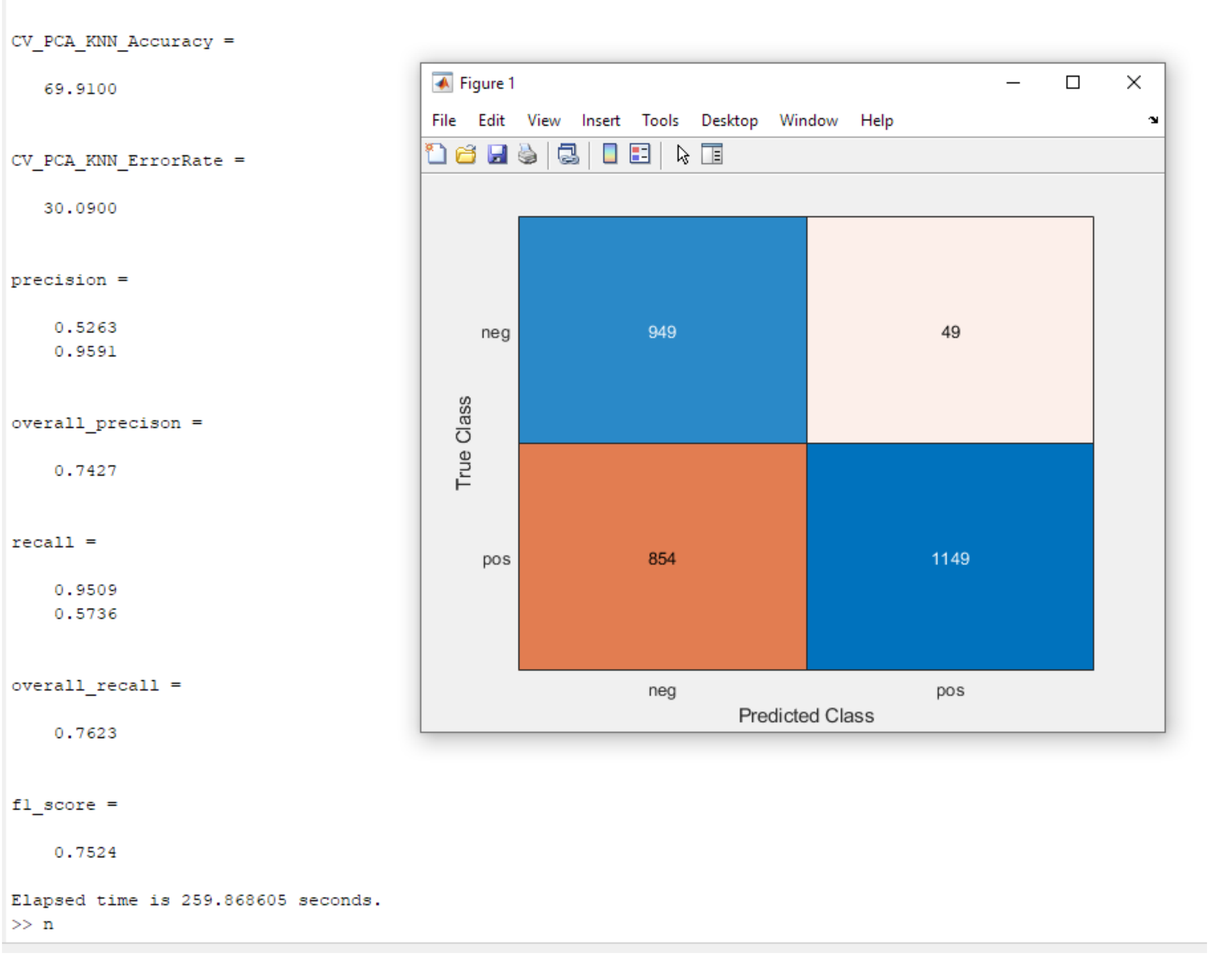
K = 1



```
CV_PCA_KNN_Accuracy =

   70.0766


CV_PCA_KNN_ErrorRate =

   29.9234


precision =

    0.5279
    0.9577


overall_precison =

    0.7428


recall =

    0.9489
    0.5771


overall_recall =

    0.7630


f1_score =

    0.7528

Elapsed time is 267.750169 seconds.
>>
```

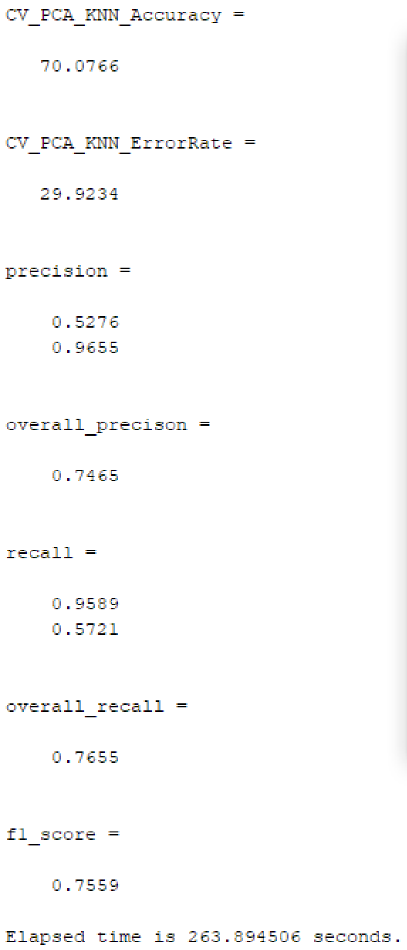
	neg	pos
neg	947	51
pos	847	1156

K = 3

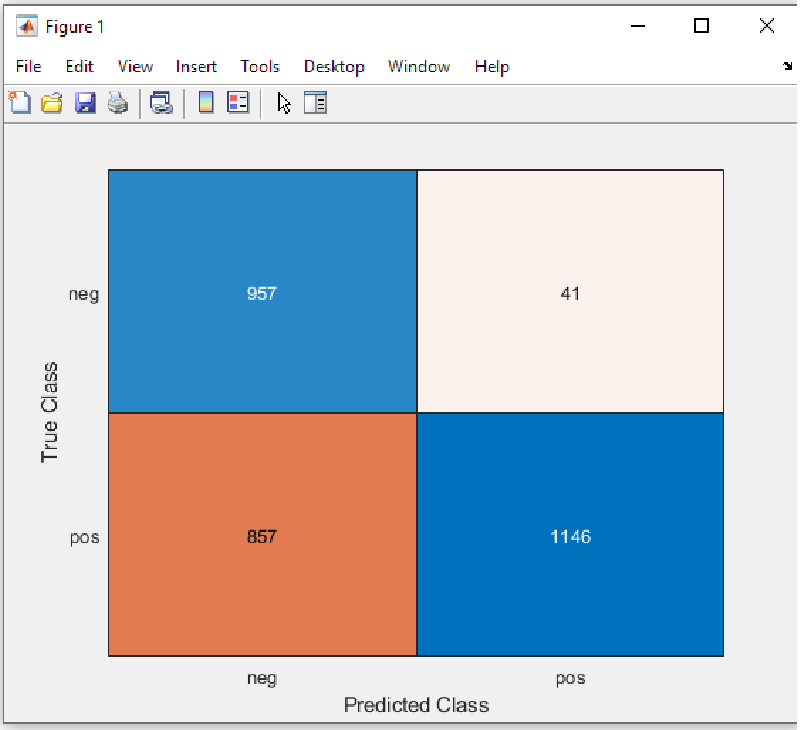


```
CV_PCA_KNN_Accuracy =

   69.9100


CV_PCA_KNN_ErrorRate =

   30.0900


precision =

    0.5263
    0.9591


overall_precison =

    0.7427


recall =

    0.9509
    0.5736


overall_recall =

    0.7623


f1_score =

    0.7524

Elapsed time is 259.868605 seconds.
>> n
```

K = 5

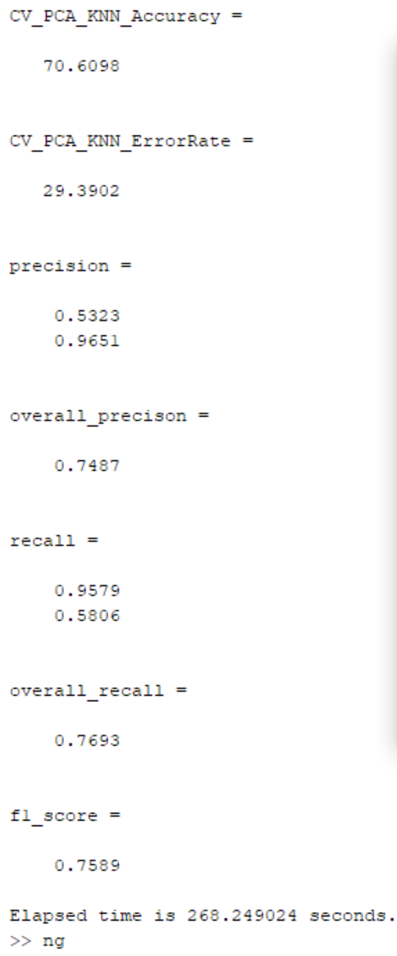


```
CV_PCA_KNN_Accuracy =

   70.0766


CV_PCA_KNN_ErrorRate =

   29.9234


precision =

    0.5276
    0.9655


overall_precison =

    0.7465


recall =

    0.9589
    0.5721


overall_recall =

    0.7655


f1_score =

    0.7559

Elapsed time is 263.894506 seconds.
```

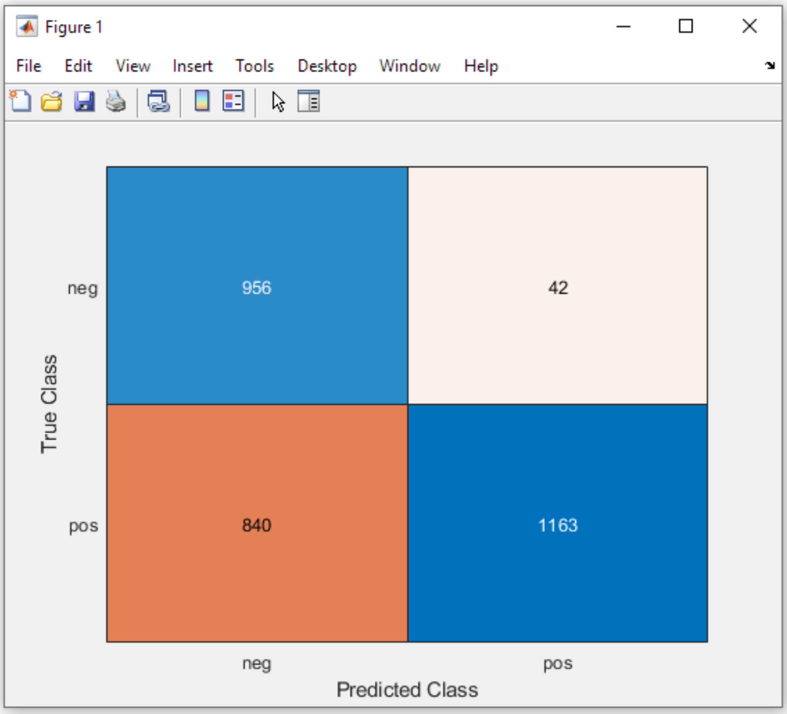


True Class \ Predicted Class	neg	pos
neg	957	41
pos	857	1146

K = 10



```
CV_PCA_KNN_Accuracy =

   70.6098


CV_PCA_KNN_ErrorRate =

   29.3902


precision =

    0.5323
    0.9651


overall_precison =

    0.7487


recall =

    0.9579
    0.5806


overall_recall =

    0.7693


f1_score =

    0.7589

Elapsed time is 268.249024 seconds.
>> ng
```



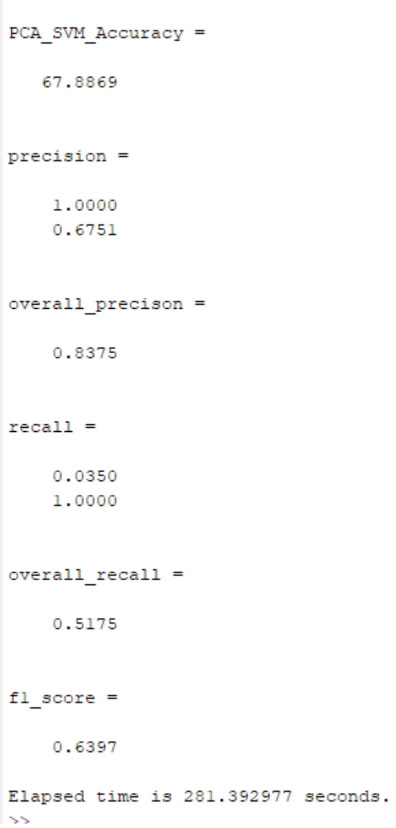
PCA SVM Classifier

For the SVM classifier, our group decided to test the gaussian and polynomial kernels.
As shown from the figures below, the gaussian and polynomial kernels both had the same accuracies in the 80:20 split and the same accuracies again in the k-fold cross validation procedure. In the case of the 80:20 training and testing split, both kernels achieved an accuracy of 67.89%. Whereas in the k-fold cross validation, both kernels were a little bit less accurate at 67.51%. In both cases, there were no false negatives.

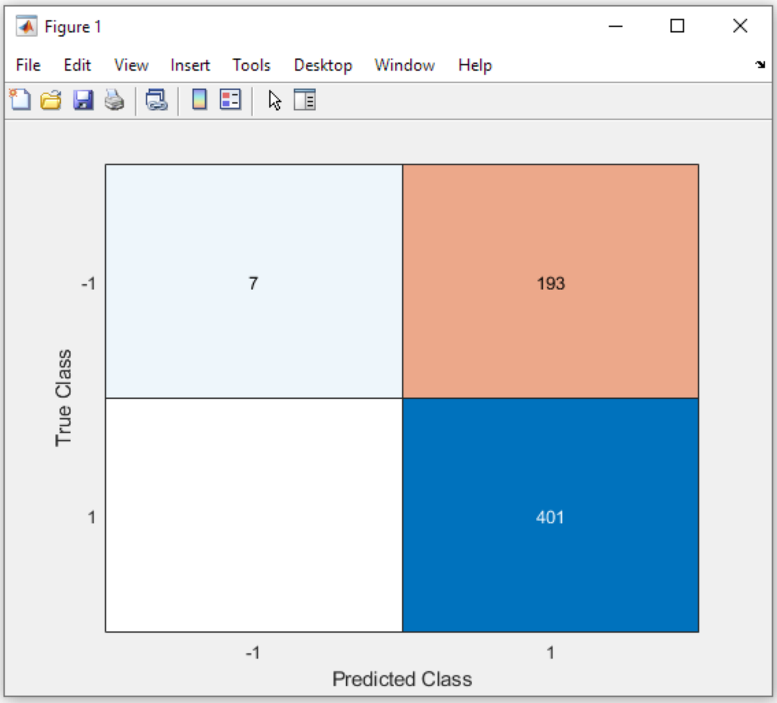
While SVM gives us a lot of flexibility through the use of different kernels, the combination of SVM and PCA gives us less accuracy than KNN. Also, SVM works better with more features or higher dimensions and smaller datasets. However, our dataset is on the larger end and due to the application of PCA, our dimensions have also been reduced. This means SVM is not the most suitable classifier for PCA in our case.

PCA SVM Classifier

Gaussian Scores and Confusion Matrix

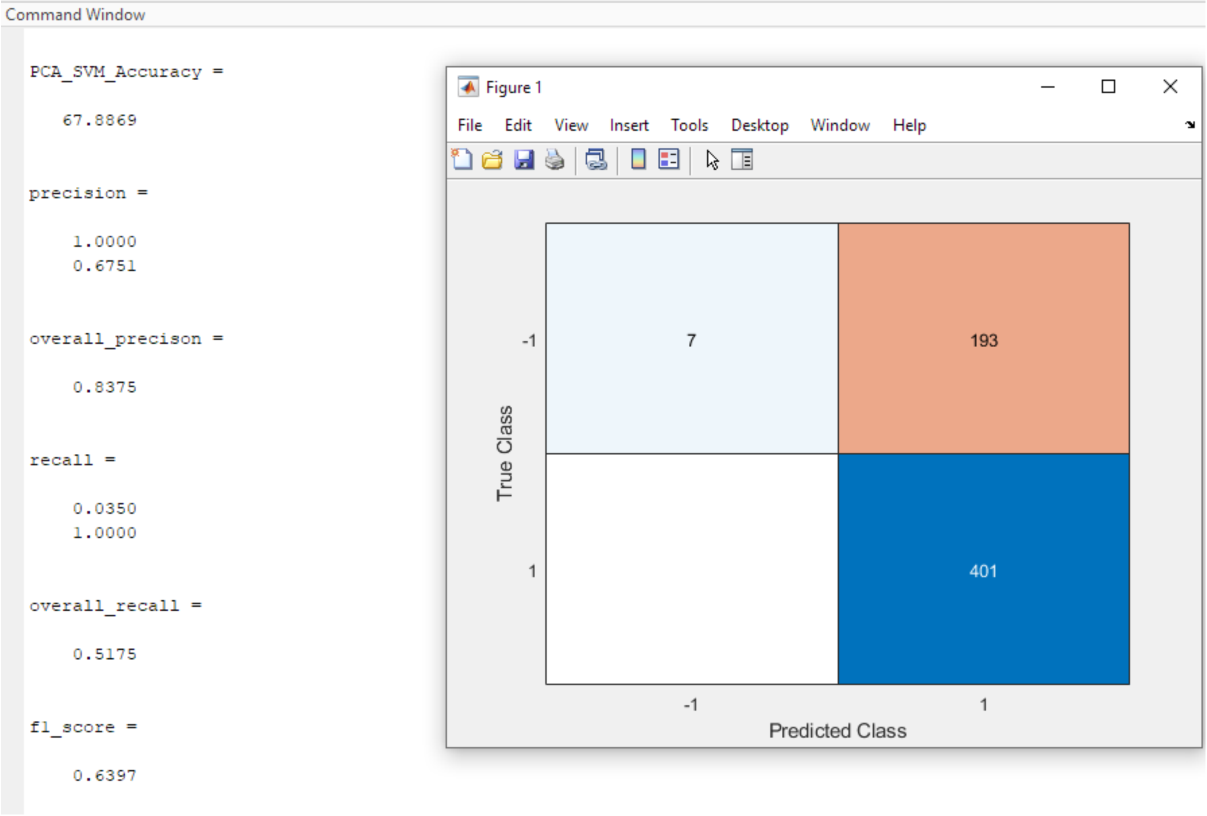


```
PCA_SVM_Accuracy =

   67.8869


precision =

    1.0000
    0.6751


overall_precison =

    0.8375


recall =

    0.0350
    1.0000


overall_recall =

    0.5175


f1_score =

    0.6397

Elapsed time is 281.392977 seconds.
>>
```



	-1	1
-1	7	193
1		401

Polynomial Scores and Confusion Matrix

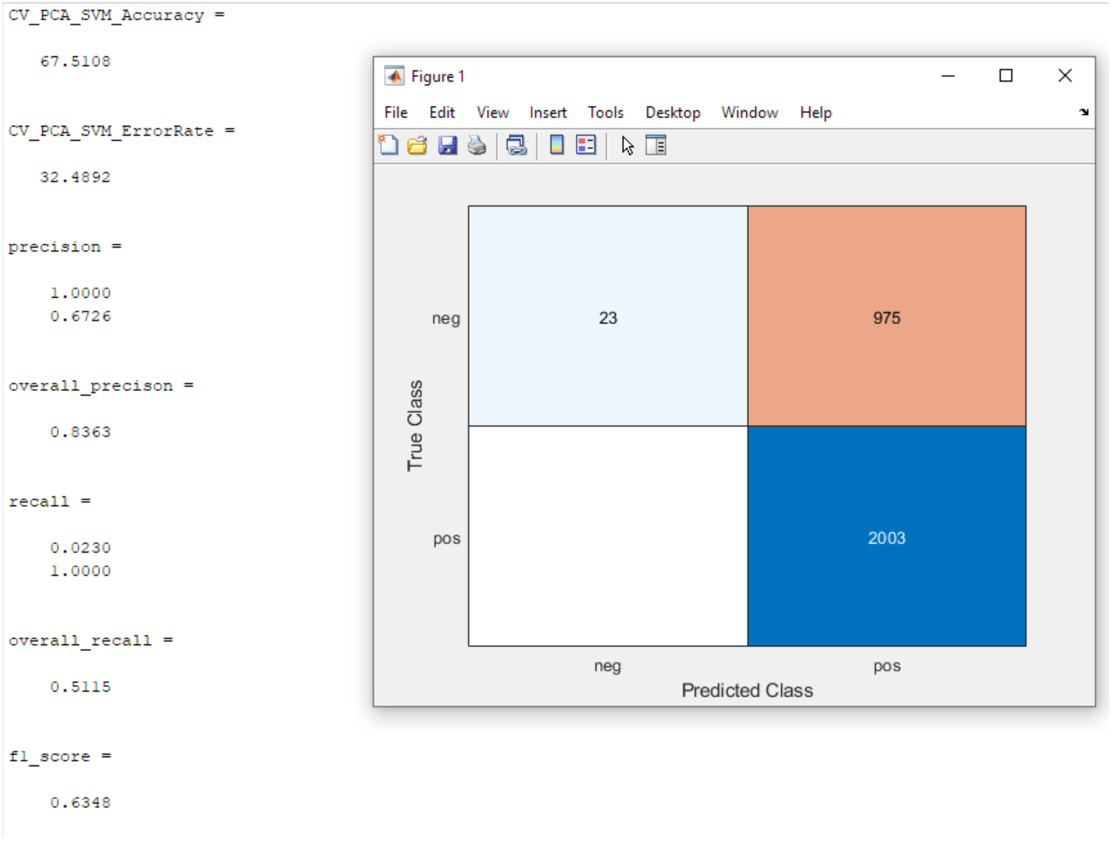


```
PCA_SVM_Accuracy =

   67.8869


precision =

    1.0000
    0.6751


overall_precison =

    0.8375


recall =

    0.0350
    1.0000


overall_recall =

    0.5175


f1_score =

    0.6397
```

True Class \ Predicted Class	-1	1
-1	7	193
1		401

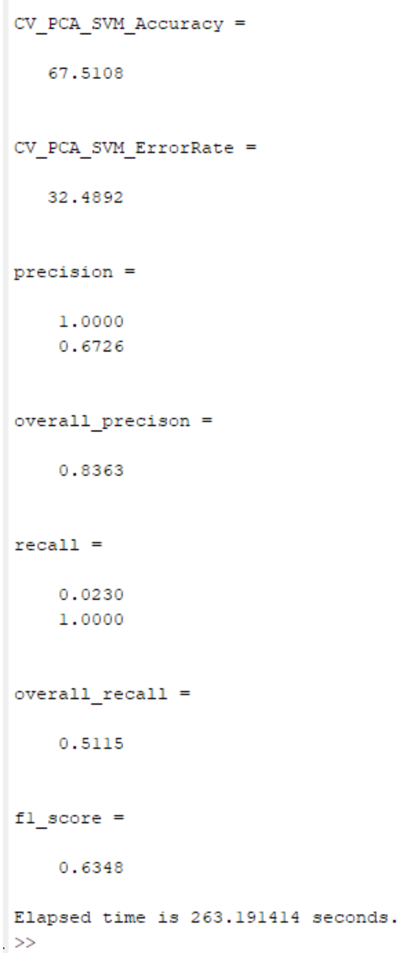
Cross Validated PCA SVM Classifier

Gaussian Scores and Confusion Matrix

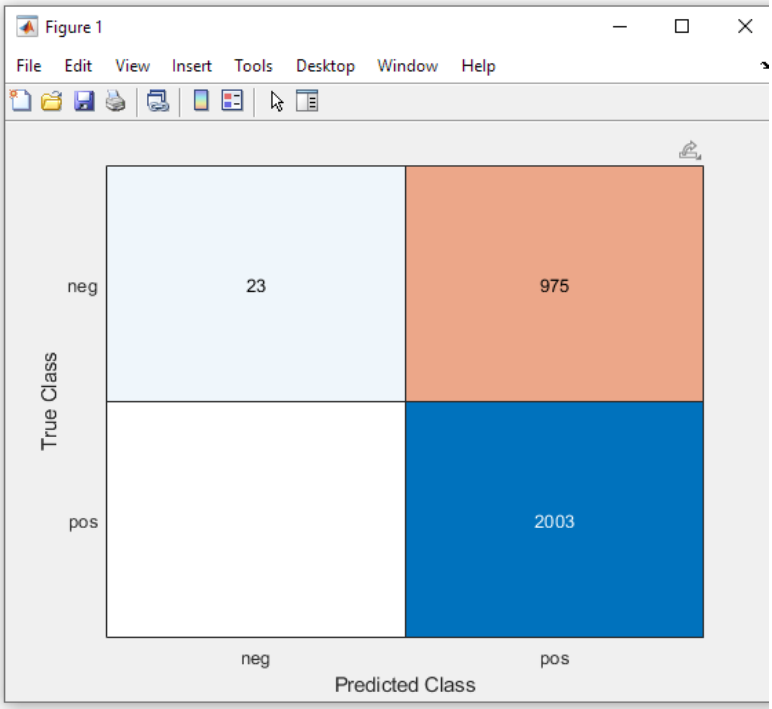


```
CV_PCA_SVM_Accuracy =

   67.5108


CV_PCA_SVM_ErrorRate =

   32.4892


precision =

    1.0000
    0.6726


overall_precison =

    0.8363


recall =

    0.0230
    1.0000


overall_recall =

    0.5115


f1_score =

    0.6348
```

Polynomial Scores and Confusion Matrix



```
CV_PCA_SVM_Accuracy =

   67.5108


CV_PCA_SVM_ErrorRate =

   32.4892


precision =

    1.0000
    0.6726


overall_precison =

    0.8363


recall =

    0.0230
    1.0000


overall_recall =

    0.5115


f1_score =

    0.6348

Elapsed time is 263.191414 seconds.
>>
```



In conclusion, the best classifier and parameter combination for PCA is the KNN classifier where K = 1 and training the model using the 80:20 split. This combination boasts a high accuracy of 72.88% and an F1 score of 0.7721.

Detection Implementation - Sliding Window

Once all feature descriptors were evaluated, we decided on using the HOG SVM Polynomial Classifier for detection. The sliding window detector runs and saves the pedestrian video. The saved video is called ‘Pedestrian_Detection.avi’ and can be found once the Sliding_Window_Detector.m has run.