Module 16: Support Vector Machines (SVMs)

Video Transcripts

Video 1: Nonlinear Features

So far, we've learned how to build classification models for datasets using k-nearest neighbors, logistic regression, and decision trees. In this module, we will add another model to that list, support vector machines or SVM. The data that we will use in this module comes from a study of the constituents of three different types of Italian wines. These include chemical compounds such as alcohol, malic acid, magnesium, phenols, and flavonoids, as well as visual characteristics such as the color intensity and the hue of the wine. In total, there are 12 attributes recorded for each of the 178 wines. And our goal will be to create a classifier that can reliably distinguish the three different classes from a few of these 12 attributes.

To begin, we load the dataset with the load_wine() method of scikit-learn .datasets. Then we store the attributes in a pandas DataFrame with the feature names as column headers. The target variable is an integer 0, 1, or 2, stored in the class column of the DataFrame. Here's the table. You can see that all of the quantities are numerical and the last column is the target integer. The data is relatively well balanced with 59 entries of class 0, 71 of class 1, and 48 of class 2. In order to visualize our results, we will use only two of the features, the total_phenols and the color_intensity.

Here's a reduced table with only those columns plus the target column. And here is the scatterplot. Clearly class 1, the red dots, is characterized by a

low color intensity, whereas class 2, the blue dots, has a higher color intensity than class 1, but a low phenol count. Class 0 also has more intense color than class 1, but has larger phenol content than class 2. Let's see whether the classification algorithms we've learned thus far can recognize these patterns as easily as we just did.

These three plots show the decision boundaries found by three different classification models. A decision tree of depth two, k-nearest neighbors with five neighbors, and multinomial logistic regression. Personally, the decision tree is the most appealing to me because it captures our intuition of the classes and it can easily be conveyed with words. Class 1 wines have low color intensity. Classes 0 and 2 are more intense. However, class 2 has lower phenol count than class 0.

The downside of the decision tree is that it is highly dependent on the max_depth parameter. Were we to increase this to say 10, we would see a very broken line. K-nearest neighbors seems to have the highest accuracy on the training data. But we also see that the decision boundaries are convoluted and make sharp turns to capture single data points. We get the sense that this model is overfitted and has high variance. The multinomial logistic regression, on the other hand, is very stable. Its main hyperparameter is the regularization weight. However, in this case, the decision boundaries are insensitive to that parameter. So what we see here is the single basic solution for multinomial logistic regression.

The big downside to logistic regression with respect to KNN is that it is limited to straight boundaries. But we've already learned about a technique for introducing nonlinearities into linear models using nonlinear features.

We saw this in the context of linear regression. Let's try it again. But this time with logistic regression. The features that we already have in the model are x_0, the phenol count, and x_1, the color intensity. These are linear features because they depend linearly on the data. Let's add quadratic features generated from x_0 and x_1. There are three possible quadratic features. Phi_0 equals the product of x_0 and x_1, then phi_1 equals to x_0^2 and phi_2 equals to x_1^2. We now create a new DataFrame called X by taking the columns for total_phenols and color_intensity from the original dataset and adding three new columns, phi0, phi1, and phi2. The resulting table has five feature columns.

We can now run multinomial logistic regression using these five features. OK this looks a little bit better than the first case. The three nonlinear features give the boundaries a slight curvature. And the resulting model is able to correctly classify a larger portion of the data points. But there are other nonlinear features that we could try. We could try cubic functions, quartic, or quintic functions. We could try general polynomials up to some high degree. We could also try exponential and logarithmic functions, as well as trigonometric functions. All of these can be tested easily with the approach we just saw, simply by adding more columns to the dataset. In this picture, we see the result from quadratic, cubic, exponential, logarithmic, trigonometric, and Fourier feature vectors. By trigonometric, I mean that we include sine and cosines of x_0 and x_1.

The Fourier basis has a larger set of trigonometric functions, sine and cosine of x, but also 2x, 3x and 4x, for both x_0 and x_1. This results in a much wavier boundary. The main point is that we can easily create curved boundaries for logistic regression by adding nonlinear features. But how

should we select these features? How do we decide between all of these options plus many others that we did not consider? Or perhaps we should combine these by taking, say, a few exponential features and a few trigonometric features. The possibilities are endless. Well, one answer to this question is to use our domain knowledge. That is, if we have some intuition about our problem, then perhaps we can translate that into some idea about which features to include.

Another answer is regularization. We've already seen how the LASSO can be used to rank the features in terms of their importance for classification. Using this technique, we can begin with a large number of features and prune it down to something manageable by increasing the regularization weight. But in the next video, we will learn a different approach to this question in which we use the kernel representation of the problem to find solutions for a large or even infinite feature list. I'll see you then.

Video 2: The Kernel Trick

In this video, we introduce the concept of a kernel. A kernel is a function that takes two data vectors as inputs and returns a number. This number gives a sense of the similarity between the two vectors. However, this notion of similarity is very abstract and there are many functions that qualify as kernels. Kernels are central to the success of support vector machines. However, they can also be used to enhance other algorithms. There are kernel-based versions of linear regression, logistic regression, and even relatively unrelated algorithms such as principal components analysis.

The main insight that leads to kernels is as follows. All of the algorithms that we have encountered thus far depend on the data. But some of these algorithms depend on the data in a very special way. They depend on how the data is arranged geometrically in space. The geometry of a dataset can be captured by computing the similarity of every pair of data points. In mathematics, the similarity between two vectors can be computed as their inner product, also known as the dot product. So any algorithm that depends only on the geometry of the dataset can be cast in terms of dot products. And if we can cast an algorithm in terms of dot products, then we can replace those with a kernel function and thus gain access to a much wider range of possibilities.

To demonstrate the technique, let's return to linear regression. In this problem, there is one input, x, and one output, y. And we have collected a dataset with three samples of x and y. Our linear model is y(x_i) is equal to β_0 plus β_1 times phi(x_i), where phi is some nonlinear feature transformation. More generally, we'd like to allow for a large number of features. And in that case, we should cast the model in a vector notation. In the scalar case, we had one feature and two coefficients, β_0 and β_1. In general, we will have M minus 1 features and M coefficients, β_0 through β_M−1. And we arrange these into a vector.

The features are also arranged into a vector with M elements beginning with a 0th feature equal to 1. Then the model takes on this very simple form, y(x_i) equals β transpose times phi(x_i). As we know, we find the parameters β by minimizing the quadratic loss function, which is the sum of the prediction errors squared. Here we are also considering an additional quadratic regularization term, which penalizes the squares of the

coefficients with a regularization strength of λ over 2. The division by two is a mathematical convenience, as we'll see. The second line of the equation plugs in the model and also expresses the regularization terms in vector notation. This is an unconstrained convex optimization problem, and it therefore has a single solution at the bottom of a bowl-shaped loss function.

This solution is characterized by the fact that the slope of the function equals zero at that location. So a way of computing the solution to linear regression is to find the slope of the loss function by taking its derivative and equating it to zero. Let's do that. The derivative of the first term produces a sum over data points of β transpose times phi(x_i) minus y_i. This is just the model error for a sample x_i. And we multiply that by phi(x_i), which is a vector of length M. This plus λ times the vector β equals zero. Call this equation 1. We can remove the remaining summation from the notation by defining a vector form for y_i and a matrix form for phi(x_i).

Let's define Phi as an N × M matrix that holds the features for each of the x_i data points as rows. We also define a column vector, Y, to hold the output data. This is the target variable, the class of wine in our example. Using these definitions, condition one becomes Phi transpose Phi times β minus Phi transpose y plus λβ equals zero. And from here we can obtain the optimal β as Phi transpose Phi, plus λ times I inverse, times Phi transpose times Y. Notice that Phi transpose Phi is an M × M matrix. So if we have three features, we need only to invert a three by three matrix. I_M here is the M-dimensional identity matrix.

The nice thing about this is that once we have computed the optimal β, we can then discard the training data and keep only the M coefficients β_0 through…at β_M−1. Then if we need to make a prediction for a new data point x_n, we simply compute the feature vector for x_n and multiply it by β.

Next, we will develop an alternative approach to linear regression that will reveal a different intuition. We begin by defining in an alternative set of parameters, α_i, as the negative of the prediction error for data point i divided by λ. Notice that in this framing, we will have one α parameter for every data point i, whereas before we had one β parameter per feature. Notice also that we have assumed that λ is not zero. So we are forced to have at least a little bit of regularization for this to work. With this definition, we can also write α_i as 1 over λ times β transpose phi(x_i) minus y_i.

Now plug this definition of α_i's into equation 1 and then divide the whole thing by λ. Then we get the following expression between optimal βs and optimal αs. That is, β must equal the sum of α_i times phi(x_i). In vector notation, that is β equals phi transpose times α, where α is the vector containing the α_i's. We should also put the definition of the αs equation 2 into this matrix form. That is negative λ times α equals Phi times β minus Y. Call this equation 4. Combining three and four to eliminate β, we get the formula shown here, which we can use to solve for the optimal α. That α equals Phi times Phi transpose, minus λ times I inverse times Y.

Let's compare the expressions for the optimal αs and optimal βs. They look very similar. Notice however, that computing α requires inverting an N × N matrix: Phi times Phi transpose minus λI. Whereas for β it was an M × M matrix. In most problems, the number of data points, N, will be much larger

than the number of features, M. And so it seems like solving this in terms of α is actually more difficult. Well, that is a good point, but let's press onward.

How do we make predictions with αs? For a new data point, x_new, we know that the prediction is found with β transpose times Phi(x_new). Using equation 3 to replace β with α, we get α transpose times Phi times phi(x_new), which expanding the notation gives us the sum over all data points of the α coefficient times phi transpose of x_i times phi(x_new). Let's compare this to the standard method for prediction with βs. The main thing to notice here is that the formula with αs involves a phi transpose of x_i term, whereas the formulas with βs does not. So in the original method with βs, once we train the coefficients, we can throw away the training data.

However, with αs, this is no longer true. Making a prediction involves all of the training data, so we have to hold on to it. Again, the method with αs seems worse than the method with βs. But don't worry, the benefits of the alternative method will soon be revealed. Let's have a closer look at the N × N matrix: Phi times Phi transpose that shows up in the alternative formulation. When we expand this out, we see that each term in the matrix is of the form phi transpose of x_i times phi(x_j). This type of operation is called a dot product and it captures a sense of the similarity between two vectors. In this case phi(x_i) and phi(x_j). This is an important matrix and we call it the kernel matrix. And denote it by a capital letter K. It captures the geometric content of the dataset projected through phi into some possibly high dimensional space.

The kernel function K(x_i) and x_j is a function that delivers each of the elements of the kernel matrix. This is denoted with a lowercase k, and it

takes two data points and returns the dot product of their feature representations. So in the alternative approach, we can replace the feature vectors phi with a single kernel function in order to compute the α parameters. What about prediction? For a given new data point x_n, the predicted output is the sum over all points of α, phi transpose x_i, times phi(x_new), which can be expressed as a weighted sum over the data points of the kernel function evaluated on the training data and the new data.

So in conclusion, the alternative approach allows us to do all of regularized linear regression, both training and prediction, using a kernel function in place of a feature vector. But it is not yet clear why we would want to do this. We will get to that in the upcoming video. I'll see you then.

Video 3: Kernel Trick Examples

To summarize where we are, we have seen two ways of introducing nonlinearity into linear regression. The feature-based approach applies a transformation, phi, to the data. And the kernel-based approach applies a kernel function, K, to every pair of data points. The models we build with feature-based approach have M coefficients β, and the models we build with the kernels have N coefficients α. The procedures for training these models are similar. For feature-based, we build a pandas DataFrame for the phi matrix and with Y as its last column. We then fit a linear regression model to this data to obtain the coefficients β_0 through β_M−1.

The kernel-based approach is very similar except that now instead of populating the table with features, we populate it with the kernel matrix. This is an N × N matrix. We can then compute the αs by applying standard linear regression to that dataset. All right, let's try this in code.

We'll fit this data using the kernelized, analyzed version of linear regression. We will use four different kernel functions. The linear, the quadratic, the polynomial, and the Gaussian kernel. The linear kernel applied to two vectors, x and z, equals x transpose z. We've been calling that the dot product of x and z. And the implementation in Python is with the dot method of NumPy. So here is the linear kernel function implemented with NumPy. This function computes the kernel matrix from a kernel function, kfunc, and a dataset, X. N is the first dimension of X.

And then K is computed by applying the kernel to each pair of row vectors in X. We can use this function to find a kernel matrix for a linear kernel. Note that we have a second term here that is 0.1 times an identity matrix of size N. This is the regularization term that we saw is needed to make this work. Now here comes the trick. We use the kernel matrix as the dataset to train a linear regression in the usual way and with Y as the target. Upon doing this, scikit-learn computes the αs and stores them as the coefficients. Here we see how to make a prediction with a kernel-based model. This function takes the regression model, the kernel function, the training data, and the test data for which we wish to make the prediction.

Again, we construct something very similar to the kernel matrix. But now, instead of evaluating the kernel function on pairs of training vectors, we do it on pairs of training and testing vectors. The result is as expected. The linear kernel function is equivalent to just using the original linear features. Let's look now at some more useful kernel functions. The quadratic kernel for vectors x and z equals (x transpose z plus 1)^2. It is not immediately obvious why this formula makes sense.

So let's look at an example in R^2. We compute the quadratic kernel for two vectors in R^2 by plugging x is equal to x_1, x_2, and z is equal to z_1, z_2 into the formula. And we get x_1, z_1 plus x_2, z_2 plus 1^2. We won't do it here, but if you expand that square, you'll see that you can equivalently write this as the product of a feature vector, phi, applied to x times phi(z). And phi(x) will contain components 1, x_1, x_2, x_1 x_2, x_1^2, and x_2^2. That is all of the monomials of the components of x up to order two. In code, all we have to do is replace the linear kernel with a quadratic kernel and the rest remains the same. The quadratic kernel function implements the formula we just saw, and we use that to build the kernel matrix. Then we feed that kernel matrix into the fit function of the linear regression. And here's the result. It is the same as you would get if you included all of the quadratic monomials as features. But a lot easier to do.

The next level up is the polynomial kernel function, which generalizes the quadratic by replacing the 2 in the formula with a positive integer, d. What features does this kernel correspond to? You guessed it. A feature vector with all monomials up to order d. That can be a pretty big feature vector. To be precise, the number of dth-order monomials equals M plus d choose d, where M is the length of x. M plus d choose d is: M plus d factorial divided by d factorial, N divided by M factorial. For M equals 2 and d equals 2, this is a modest six features. But for M is equal to 10 dimensions, and polynomial orders of d equals 10, we get a feature vector with over 184,000 items. And this is where the kernel method really begins to shine. So, so far, we've been building our own kernel functions. But scikit-learn's metrics.pairwise package provides some of the most common ones out of the box. Here I'm using scikit-learn's polynomial_kernel function and passing a degree value of 3.

The kernel matrix that we get is equivalent to a feature vector with all monomials up to degree three.

Finally, let's have a look at another use for kernel, the Gaussian kernel. The formula for the Gaussian kernel function is the exponential of negative gamma times the square of the norm of x minus c. Here gamma is a hyperparameter. This kernel is also known as a radial basis function, since it depends only on the distance between two points. As a similarity measure, it is saying that points are similar insofar as they are close together in space. That is pretty reasonable, but it is not obvious how this can be expressed as a dot product of two feature vectors. And here's where it gets strange. The feature vector that this kernel corresponds to has infinitely many entries. It is obtained by a process of appending monomials of higher and higher order in a way that approaches the Gaussian kernel, the more monomials you add. This is of course impossible to do in features face, but with kernels, it turns out to be very simple.

To apply the Gaussian kernel in code, all we need to do is use the rbf_kernel function in place of the polynomial kernel function. The Gaussian kernel often produces pretty good results, as we can see here. The kernel trick can be applied to other algorithms in addition to linear regression, including principal components analysis and logistic regression. In the next video, we will learn about a model for which the kernel trick is especially beneficial: The maximum margin classifier. I'll see you then.

Video 4: Maximum Margin Classifier

In this video, we will learn a new technique called the maximum margin classifier, which like logistic regression, produces linear boundaries and is amenable to the kernel trick. Here's the motivation for the maximum margin classifier. Have a look at this dataset. It has two classes and two features. Let's say we want to build a linear classifier for it. Well, there are many possibilities. For example, this red line is one and the green line is another. They are equivalent in the sense that they both achieve 100% accuracy on the training data. But which one would you prefer? The green line or the red line? And why? Take a moment to think about that. The red line is preferable because it is more robust. Say we now draw a test data point that is of the green class. The data point is more likely to fall on the wrong side of the green line than it is of the red line. This is because the green line comes closer to the training dataset.

And because new data points are more likely to appear near the training data, it is reasonable to draw our decision boundary as far from the training data as possible. To formalize this notion, we introduce the concept of the margin. The margin of a decision boundary is the perpendicular distance from the boundary to the nearest data point in the training set. Amongst all of the candidate decision boundaries, we should choose the one that, like the red line, maximizes the margin.

Next, we will express these perpendicular distances in terms of our model variables. But first, let's return to our previous setup, in nonlinear feature space, with features Φ_1 through Φ_M minus 1. The linear decision boundary is given by the formula $y(x) = \beta_0 + \beta_1 \Phi_1(x) + \beta_2 \Phi_2(x)$ all the

way to β_M minus 1, Φ_M minus 1(x). For this part, it will be convenient to adjust our vector notation a bit. We will remove the 1 from the top of the feature vector and also the β_0 from the top of the coefficients vector. And so, in vector form, the model becomes y = β_0 + β^T times Φ(x). The reason for this redefinition is that it gives β a very nice interpretation as the vector that is perpendicular to the decision boundary. Here's a quick example in case you need convincing of this fact.

Consider a decision boundary that intersects the x_1 and x_2 axes at points 1 and 2, respectively. The formula for this line is β_0 + β_1 times x_1 + β_2 x_2 = 0. Plugging the two green dots into this formula, we get that β_0 + β_1 must equal 0 and β_0 + 2β_2 must equal 0, which in turn implies that β_1 equals to 2β_2. Plugging this relationship into the coefficients vector, we find that β lies in the direction two one. And we can see from similarity of the magenta and blue triangles that this direction is perpendicular to the decision boundary. Alright, let's proceed. The situation is easier to visualize in one dimension instead of two. So, let's assume that we have only a single feature, Φ_1 and two classes. The green class is encoded as plus 1 and the orange class is encoded as negative 1. The linear model y(x) = β_1 Φ(x) + β_0 appears as a straight line in this diagram. And the decision boundary is the point where y is equal to 0. The slope of the red line is β_1. And the margin is the horizontal distance from the decision point to the nearest data point, in this case an orange point.

Our goal is, given this data, to find the red line that maximizes the margin. Here's the procedure we will follow to find a good red line. First, to exclude bad solutions, we will extend boundaries from the orange data up and from the green data down. And we will prohibit the red line from going through

these boundaries. We can express these constraints mathematically by requiring that the value of the red line be greater than 1 for green data points and less than negative 1 for orange data points. These two conditions can be collapsed into a single one by multiplying them by y_i, since y_i is 1 for green dots and negative 1 for orange dots. Then we observe that amongst all of the lines that satisfy these constraints, such as these blue lines, the one that maximizes the margin is the one with the shallowest slope; the red line in this figure.

But what if we had the opposite situation where the positive green points work to the left of the negative orange points? In that case, instead of minimizing a positive β_1, we would want to maximize a negative β_1. We can capture both cases by requiring that we minimize the absolute value of β_1. That is, we want to find the shallowest slope, whether it is going up or down. To generalize this to many dimensions, all we need to do, is replace the absolute value of β_1 with the norm of the β vector. We minimize the square of this because it turns out to be easier to solve numerically and it does not affect the result. This is the optimization problem for the maximum margin classifier. And it turns out to be convex. And convex problems are great because they can be solved very efficiently with gradient descent methods.

Here's an example of maximum margin classifier computed with scikit-learn's SVC class. We'll see later how to do this in code, but what I want you to notice here is that the margin is determined by just three of the data points. We call these the support vectors. All of the other data points play no role in the model. We could move these other points around and it would make no difference, as long as none of those points enters the margin and

we keep the support vectors fixed, then the model is insensitive to variations in the data. This turns out to be a very nice feature. As we saw previously with the kernel method, the solutions to linear regression depends on all of the training data simultaneously. This is also true of logistic regression. However, for maximum margin classifiers, the result depends only on a small number of support vectors. And this fact makes this model much more amenable to the kernel trick.

There is still one problem, though. The derivation of the maximum margin classifier that we just developed, depended crucially on the green and orange data points not overlapping. If they did overlap, as in the case shown here, then the rod constraints would have precluded all solutions. To fix this problem, we need to allow the orange and green boundaries to be moved. In this particular situation, we need to move the overlapping green rod down by an amount C_i. That would open up space for the red line to get through. The position of the green rod then becomes 1 minus C_i. This is expressed in our optimization problem by replacing the 1 on the right-hand side of the constraint with a 1 minus C_i. But we want to discourage the displacement of the rods, so we penalize it in the cost function by adding a term for the total displacements, that is the sum of Cs.

Now we have two competing factors to balance, maximizing the margin and minimizing the shifting of the constraints. The relative importance of these two criteria is determined by a hyperparameter, C. A nice thing about this relaxation of the problem is that it does not destroy convexity. The problem is now bigger in the sense that there are more decision variables to compute, but it is still convex and relatively easy to solve. OK, so that's the maximum margin classifier. In the next video, we will see how the kernel

trick applied to the maximum margin classifier gives us a very powerful new model — the support vector machine. I'll see you then.

Video 5: Support Vector Machines

In previous videos, we learned that algorithms that depend only on the geometry of the data can be formulated in terms of a kernel function. Although there are advantages to using kernels for linear regression, there are also a couple of significant disadvantages. First, training can be more difficult when the kernel matrix is large. Second, predicting the response for a new vector requires applying the kernel function to each of the training datapoints; and this can be computationally expensive. And although we didn't show it, the same is true of logistic regression. We also learned of another classifier called the maximum margin classifier, for which the solution depends only on a reduced number of datapoints, which we called the support vectors.

In the case of non-overlapping class distributions, the support vectors are just the points that touch the outer edges of the margin. And in this case, there is no misclassification. In the case that the clouds do overlap, then we must relax the constraints and allow for some points to be misclassified. This picture shows the situation in 1D. The red dot is the decision boundary found by the maximum margin classifier. Only one point has been met misclassified; the green dot to the left of the boundary. We can tell that it is misclassified by noting that it has a value of C that is greater than one. The green dot to its right is correctly classified, but it also falls within the margin. Its C value is between 0 and 1. All of the rest of the datapoints have C equal to 0, indicating that they are correctly classified and outside of the margin. The maximum margin classifier is sensitive to those points that are

either touching the edge of the margin or that have a C_i value greater than 0. We call these the support vectors.

The support vector machine is nothing more than the kernel trick applied to the maximum margin classifier. This combination has been found to work very well. It preserves the flexibility of kernels, but also has good numerical properties because of the relative sparsity of the maximum margin classifier. Scikit-learn makes it very easy to construct support vector machines. Let's have a look. We will start with this example of a double spiral and see whether we can separate the two arms with the support vector machine. To build a support vector classifier, we use the SVC class of the scikit-learn .svm package. Building a support vector machine in scikit-learn is just as simple as any of the other models we've seen. We pass the parameters of the class constructor and then provide the training data to the fit method. In this case, we are invoking a linear kernel and this results in a linear decision boundary. Once the model is trained, we can evaluate new datapoints with the predict method.

Let's look at a polynomial kernel. In this case, we need to specify the three hyperparameters of the model. Gamma here is a kernel coefficient. You can set it to a positive number but omitting it or setting it to scale prompts scikit-learn to give it a good default value. r is the offset value in the polynomial kernel and it is assigned to the parameter coef0 and the SVC constructor. Unfortunately, scikit-learn currently sets this parameter to 0 if it is not explicitly assigned; and this is not what we want. So please remember to give it some positive value. Finally, the most important parameter is the degree of the polynomial, d. And this is assigned through the degree argument. And here we see the decision boundary computed with a

polynomial kernel of degree eight. Here we see the result with a Gaussian kernel, also known as a radial basis function, or RBF, with gamma set to 10. We can tune the gamma of the Gaussian kernel by varying it over some range of values and looking at the cross-validated score. Here we see that we can achieve a very high accuracy with gamma set to about 20. The best solution in fact, is achieved with gamma equal 18.7. And here we see the result.

So now let's go back to the problem we began with, that of guessing the class of a wine from its total phenol count and its color intensity. The support vector machine does quite well in producing boundaries that capture a large portion of the datapoints without being too wavy. This solution with two features has a cross-validated accuracy of 89%. This is similar to what we obtained earlier, but much simpler to do, since we did not have to build the features by hand. Setting the kernel took care of all of that for us. This simplicity, in addition to its computational efficiency, are factors that have made support vector machines one of the most popular methods for both classification and regression.

Video 6: Targeting

Companies and organizations typically have limited resources available to accomplish any given task. Thus, organizations are constantly looking for ways to innovate and accomplish things more efficiently. As you've no doubt seen in other examples in this course, predictive models can be helpful in these respects. Today I want to talk about one way in which researchers have used machine learning tools to leverage publicly-available data from the website yelp.com to help government health inspectors target restaurants that violate food safety and hygiene standards. Health

inspections in most U.S. cities are done randomly. And thus, inspectors may be wasting scarce time inspecting restaurants that have been following the rules closely and they may be missing opportunities to improve health and hygiene at places with more pressing food safety issues.

At the same time, millions of people cycle through and post yelp reviews about their experiences at these very same restaurants. The information in these reviews has the potential to improve the city's inspection efforts and could transform the way that inspections are targeted.

The researchers propose that online reviews written by the very citizens who have visited these restaurants can serve as a proxy for predicting the likely outcome of the health inspection of any given restaurant. Such a predictive model can complement the current inspection system by enlightening the Department of Health to make a more informed decision when they allocate their inspectors. The researchers collected various review and restaurant data, including ZIP code, and inspection history, and unigram and bigram text features. They then used this feature space within a support vector machine learning model. They formulated their prediction task as a classification problem and considered restaurants with zero violations as hygienic and restaurants with many violations as unhygienic. The researchers then used L1 regularization and tenfold cross-validation to avoid overfitting their model.

The learned model achieves over 82% accuracy and discriminating severe offenders from places with no violations. The model also provides insights into salient features and reviews that are indicative of the restaurants' sanitary conditions. Many of the observable characteristics that the

researchers use have strong predictive power. For example, the location, or cuisine, or inspection history, all strongly predict violations. However, the most effective predictors in the model are the textual content from the reviews themselves. For example, hygiene-related words, such as gross or sticky, are overwhelmingly negative and strongly predictive. In the future, health inspectors can use this model to better allocate inspectors to restaurants that are likely, based on the results of the model, to be violators. This is one example where prediction can be used to more efficiently allocate scarce resources in an organization. And I imagine you can come up with a few more examples from your own work experience.